ASYNCHRONOUS MIPS PROCESSORS: EDUCATIONAL SIMULATIONS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Robert Webb

July 2010

COMMITTEE MEMBERSHIP

TITLE:	Asynchronous MIPS Processors: Educational Simulations
AUTHOR:	Robert Webb
DATE SUBMITTED:	July 2010
COMMITTEE CHAIR:	Chris Lupo, Ph.D.
COMMITTEE MEMBER:	Christopher Clark, Ph.D.
COMMITTEE MEMBER:	Phillip Nico, Ph.D.

Abstract

Asynchronous MIPS Processors: Educational Simulations

Robert Webb

The system clock has been omnipresent in most mainstream chip designs. While simplifying many design problems the clock has caused the problems of clock skew, high power consumption, electromagnetic interference, and worst-case performance. In recent years, as the timing constraints of synchronous designs have been squeezed ever tighter, the efficiencies of asynchronous designs have become more attractive. By removing the clock, these issues can be mitigated. However, asynchronous designs are generally more complex and difficult to debug. In this paper I discuss the advantages of asynchronous processors and the specifics of some asynchronous designs, outline the roadblocks to asynchronous processor design, and propose a series of asynchronous designs to be used by students in tandem with traditional synchronous designs when taking an undergraduate computer architecture course.

Acknowledgements

Thank you to my parents...[1]

[1] ...for paying my tuition, among other things.

Contents

List of Figures

Chapter 1

Introduction

Computer chip designers have long relied on a system wide clock to signal the change of a state. This has resulted in designs where all instructions are assumed to introduce the worst possible delay. This bottleneck has been mitigated by solutions such as pipelining, branch prediction, and dynamic scheduling. However, the recent slowdown in the progress of designers to further increase the clock rates of chips has led to renewed interest in designing a processor without a clock.

Using an internal clock requires that a chip can only progress as fast as the slowest component, which has created increasingly complex pipelines. Even with these very complex pipelines, much of the chip is inactive for a large portion of the time between cycles. Also, as clock rates increase, the size of the circuits has become an issue because of the need for the clock pulse to reach every component of the chip before the next pulse occurs.

Asynchronous design is a possible solution to many of these roadblocks. This thesis presents the basic theory of asynchronous circuit design. To further demonstrate these ideas, simulations of asynchronous processors are introduced. The simulations are designed in parallel with popular synchronous designs to facilitate their use as pedagogical tools, as the lack of academic resources has been one of the principal reasons asynchronous designs have not reached more widespread awareness.

The available literature does not contain a small simulation of a working asynchronous processor with laboratory exercises. The primary contribution of this paper is to fill that gap. The simulations detailed here are small and easily accessible. The first could be introduced in a few additional lectures once a single cycle synchronous design has been discussed. Depending on the detail desired, introducing the pipelined design to a class could take anywhere from a few days to a few weeks of lecture and lab.

1.1 Advantages of Asynchronous Design

The advantages of asynchronous designs can be categorized primarily in terms of speed increase, elimination of the clock skew problem, reduction of electromagnetic interference, and lower power consumption.

1.1.1 Potential Speed Increase

A processor that does not have a clock can proceed when processing is complete. There is no need to wait for the next synchronizing clock cycle. For example, when using a simple cascading adder, the addition of two small numbers is fast. However, the subtraction of two small numbers takes quite a bit of time because the two's complement representation of negative numbers requires that all of the significant bits be determined in a cascade. The ALU pipeline stage must be long enough for the worst possible case, but an asynchronous processor can move on once the calculation is completed.

1.1.2 Elimination of Clock Skew

As clock rates have become faster the speed of signals across the distance of the chip has become a problem. The maximum delay of the clock signal across the chip is the clock skew, and this delay must be small when compared

to the frequency of the clock. In modern designs methods of clock distribution have become increasingly complicated and expensive in order to mitigate this problem.[12] Creating processors without a clock eliminates this problem.

1.1.3 Electromagnetic Interference

The clock pulse must be powerful enough to reach every part of the chip to synchronize operations. This powerful electric current traveling through the chip can cause significant electromagnetic interference for other nearby components. Eliminating the clock significantly reduces electromagnetic interference produced by processors because the total current will be lower and it will not pulse at a constant frequency.[12]

1.1.4 Reduced Power Consumption

Am important commercial interest in asynchronous processors has been their extremely low power consumption. The largest power requirement of a processor is the clock signal, which must occur even when the processor is idle [12]. An asynchronous processor never has to generate a clock signal, so uses less power when running. Also, upon a cache miss or other event requiring the processor to wait for other components of the system, the processor can maintain an idle state with essentially no power consumption. Because of these factors, an asynchronous design consumes significantly less power than synchronous processors. This makes an asynchronous processor an ideal candidate for a battery powered mobile device.

1.2 Roadblocks

There are several roadblocks to producing asynchronous processors. These stem from the increased difficulty of asynchronous design and the inertia of industry and educational institutions in the direction of synchronous designs. Specif-

ically, there are few design tools, trained engineers, or academic courses on the subject.

1.2.1 Design Difficulties

The lack of a synchronizing clock makes design much more difficult. Every part of the processor is acting independently, which can lead to unexpected interactions and behavior. Fixing the problems with one set of interacting systems can cause cascading problems down the pipeline.

1.2.2 Design Tools

The few design tools that are available to engineers have not been widely adopted. Also, when compared to existing synchronous tools, the asynchronous design tools automate fewer parts of the design process and do not provide as much framework for debugging and testing. For more information on available tools see [1].

1.2.3 Lack of Academic Courses

Very few schools offer any kind of academic training in asynchronous design. This has led to most engineers being unaware of the theory that has been developed by research groups developing asynchronous designs. Because the theories have not been widely taught, asynchronous design problems can seem more daunting [12].

1.3 Outline: Parallels to Computer Organization and Design

The purpose of this paper is to propose a way of integrating asynchronous design into a traditional undergraduate computer architecture class using Patterson and Hennessy's "Computer Organization and Design" (COD) [11] as a starting point. The project began with a term paper at Cal Poly in Winter 2009 with Dr. John Seng. It was impractical to learn all the theory of asynchronous design in the timespan of the 10 week quarter, so the goal was to implement a basic single cycle asynchronous processor using mostly synchronous design principals from COD while borrowing asynchronous principals only when necessary. In the end, only dual rail encoding and basic dual rail gates (defined in section 3.1) were required. However, several timing assumptions were required to make the simulation simple enough to complete in the time allotted. The result was a design that was not robust.

To expand the term project into a master's thesis, pipelining and the necessary theory to develop a robust design would be added. In order to integrate the work into existing courses, the design would continue to parallel COD. The final product would be a set of asynchronous simulations that could be used in a traditional course focused on synchronous designs to demonstrate asynchronous designs.

To parallel the machine arithmetic chapter of COD, two asynchronous ALU units are presented. The first is naive, but easy to understand, and presented in section 3.2.2. The second requires more theory, but is much more robust in that it has fewer timing assumptions, and the timing assumptions are mostly local in nature. The theory required to understand the second ALU is presented in section 4.1. The ALU design is presented in section 4.2.2.

To parallel the single cycle data-path chapter of COD, a simple asynchronous processor is presented. This design was created using the design from the text and converting the data path wires to dual-rail encoding and any logic involving

these wires to dual-rail gates. These busses are then used to create completion signals which take the place of the clock. As noted above, to make this simulation work, several naive assumptions needed to be made. This design is the subject of chapter 3.

To parallel the pipelining chapter of COD, a pipelined processor using asynchronous design techniques is presented. This is presented in chapter 4. The chapter includes a detailed description of the theory that was incorporated into the design in section 4.1. The design of the simulation is detailed in section 4.2.

This report is concluded with a description of the educational resources that accompany the project in chapter 5 and appendix D and a discussion of conclusions and possible future research in chapter 6. Also included as appendices are the complete source code for the single-cycle synchronous processor (appendix A), single-cycle asynchronous processor (appendix B), pipelined asynchronous processor (appendix C).

Chapter 2

Previous Work

2.1 Asynchronous Logic

For small scale circuits the cumulative effects of gate and wire delay are not difficult to deal with. For this reason, many small circuits do not have a clock because one is simply not needed. The fact that these circuits were asynchronous is generally not explicitly stated. For a discussion of these circuits, see chapter 8 of [14].

When the number of switches or transistors in a circuit began to grow exponentially, quickly approaching the millions, the problem of synchronizing the behavior of a circuit became nontrivial. This was exacerbated by the problem that these pioneers were literally manufacturing the first computers, so all of these circuits had to be laid out by hand on paper. Because CAD simulations were not available, the solution that most designers used was to determine an upper bound on the worst case delay of a circuit or segment of a circuit and use an external clock to signal the completion of the logic and stability of the outputs.

2.2 Early Developments

An exception to this trend toward synchronous designs were the designers of ILLIAC II at the University of Illinois in the late 1950's and early 1960's [10]. One of the first pipelined computers, it also had many components designed to work without an external clock. Much of the early theory of large scale asynchronous circuits was developed by the designers of ILLIAC II, specifically David E. Muller. Because, at the time, wire delay was insignificant compared to gate delay, a theory of "Speed-Independent" (SI) circuits was developed to generate logic that would not glitch given arbitrary gate delay. A SI circuit is assumed to have no wire delay. A new gate was developed to accomplish this, the Muller C-Element [10]. These developments are discussed in detail in section 4.1. The handshaking protocols used in the pipelined simulation were also developed in this period by Muller. This remained the standard handshaking protocol until 1988 when Ivan Sutherland proposed an improvement in [13], but this improvement is not used here because of the intended audience and additional circuit complexity.

2.3 Modern Developments

The principal inspiration for this project has been the work of the Asynchronous Processor Group at the California Institute of Technology, led by Alain J. Martin. One of the early contributions of this research group was the paper that described the limitations of the successor to SI circuit theory, "Delay-Insensitive" (DI) circuit theory.

As manufacturing processes improved to include more complex circuits with faster switching, it became clear that wire delay and gate delay were concurrent problems. This led to the development a theory of DI circuits where the delay of wires AND gates could be any arbitrary non-negative value without causing any unstable outputs. In the early 80's, Martin developed a model for circuit timing similar to the context-free languages used in computability theory and language theory to prove that any circuit conforming to the requirements of DI would only

be functionally equivalent to a circuit made entirely from C-elements. As the C-element is not a universal gate[1], this creates a very limited set of potential DI circuits [7].

The limitations led to the development of a compromise to the strict requirements of DI. Called "Quasi-Delay Insensitive" (QDI), nearly all asynchronous development has followed this paradigm. The compromise is to simply allow a wire to fork out locally and guarantee that the data carried on that wire will reach all the local receivers at the same time. These are called "iso-chronic" forks. Manohar and Martin proved that these circuits are Turing-complete in 1995. [2] Because of the limitations of the original definition of DI and the adoption of QDI, most modern references to DI are, in fact, referring to the assumptions and requirements of QDI.

In the 80's, Cal Tech also developed the first QDI general purpose processor, mostly as a proof-of-concept. A RISC[2] design, it was finely pipelined and employed many opportunities for concurrent processing that would have been significantly more difficult in a synchronous design. Most significantly, the execute, memory, and write back stages of a traditional 5 stage pipeline were designed to run in parallel, asynchronously. Specifically, there were several busses to write back into the register file, and different instructions could simultaneously be accessing the ALU and memory modules, while asynchronously writing back to the registers. Results were promising, even with manufacturing defects. Most impressively, the processor was entirely QDI. This allowed it to run at a wide range of voltages and temperatures [3, 4].

The advancements in asynchronous design led to speculation that clockless processors were going to be much more widely used in the 90's [5]. While this speculation proved incorrect, progress in asynchronous design continued. Martin published the design of a fast asynchronous adder in 1991 [6]. This design was optimized by going down to the transistor level, so was not used in this project as the intended audience for the simulation is computer science students, not

[1] C-elements cannot be used to create a circuit that is equivalent to all the other standard logic gates.

[2] Reduced Instruction Set Computing

electrical engineers, however, it could significantly increase the logic speed of asynchronous circuits. For a discussion of several asynchronous adders, including the one detailed in section 4.2.2 and the one designed by Martin, see section 5.3 of [12].

As an additional proof-of-concept, Martin's group produced an asynchronous processor running the MIPS R3000 machine language [8, 9]. This processor is currently the state of the art in terms of asynchronous design and the next step in the logical development after [3]. The papers also developed a new metric for the performance of asynchronous processors, $E\tau^2$ where E is the average energy per instruction and τ is the average instruction execution time. Experiments suggested that, for a given design, that $E\tau^2$ is roughly constant because voltage varies inversely with the square of speed.

2.4 Educational Resources

As asynchronous design has been "the next big thing" and "just around the corner" for quite some time, it has been only recently that the subject has been the subject of publications beyond short journal articles or technical reports. The lack of textbooks has been a roadblock to the teaching of asynchronous design because most available material was not very student-friendly. Two monographs have been written to remedy this. Davis and Nowick wrote a circuit design manual in 1997 is targeted more toward the experienced designer [1]. Sparsøwrote a textbook targeted to graduate students and advanced undergraduates in 2006 that is freely available on his web page [12]. This textbook has been invaluable to this project. It was the primary reference, especially for pipeline design, even though the principle pipeline control design dates back to Muller and Illiac II. As noted in section 1, these books do not include a small, student accessible, simulation of an asynchronous processor.

Chapter 3

A Single Cycle Asynchronous MIPS Processor

This chapter describes a single cycle asynchronous MIPS processor that closely follows the designs presented in COD. To make the simulation accessible, asynchronous circuit theory is used as little as possible. The result is that the processor is simple, but not robust.

3.1 Design Principles

3.1.1 Dual Rail Encoding

In dual rail encoding every bit A is represented by a pair of bits AH and AL, for A-high and A-low. If $A = 0$ then $AH = 0$ and $AL = 1$. If $A = 1$ then $AH = 1$ and $AL = 0$. If A has not been fully determined, then $AH = 0$ and $AL = 0$. The situation where $AH = AL = 1$ has different meanings in each of the simulations. For the first simulation, this case will also imply that the bit is undetermined. For the more advanced pipeline simulation, the circuit will be designed so that this case cannot occur. See section 4.1.4 for more on this more

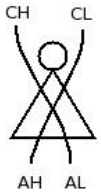

Figure 3.1: basic dual rail NOT gate: NOT2

advanced paradigm.

3.1.2 Dual Rail Logic

All the logic gates used in the traditional encoding of data have an equivalent gate that takes dual rail data as input and produces dual rail data as output.

$$\overline{(AH, AL)} = (AL, AH)$$

so $\overline{(1,0)} = (0,1)$ and $\overline{(0,1)} = (1,0)$ as shown in figure 3.1. If A has not yet been determined then $\overline{A}$ will not be determined either

$$\overline{(0,0)} = (0,0).$$

Similarly

$$(AH, AL) \wedge (BH, BL) = (AH \wedge BH, AL \vee BL)$$

as shown in figure 3.2 and

$$(AH, AL) \vee (BH, BL) = (AH \vee BH, AL \wedge BL)$$

as shown in figure 3.3.

These new gates are referred to as NOT2, AND2, and OR2 in the simulations using these constructions.

As can be seen, if both inputs to either AND2 or OR2 are undetermined then the output is undetermined. If one of the inputs to AND2 is low then it will output low, even if the other is undetermined. If one of the inputs to AND2 is high then it will output undetermined if the other input is undetermined. If

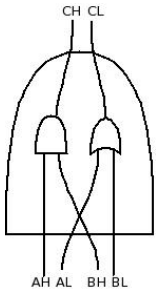

Figure 3.2: basic dual rail AND gate: AND2

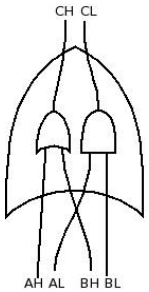

Figure 3.3: basic dual rail OR gate: OR2

one of the inputs to OR2 is high then it will output high, even if the other is undetermined. If one of the inputs to OR2 is low then it will output undetermined if the other input is undetermined.

The fact that these new "gates" are actually more complicated circuits constructed from traditional gates implies that timing is still a concern. For example, if the AND gate inside the AND2 is slower than the OR gate inside AND2, then the output of the AND2 will produce an output of $(1, 1)$ when transitioning from an output of $(1, 0)$ to an output of $(0, 1)$. This is an example of a "glitch". Because of this problem an output of $(1, 1)$ is defined as undetermined, and XOR is used to determine if the output of a circuit using these gates is ready to be used. While this method is used in the first simulation, a more sophisticated construction is used in the second simulation that will keep these "glitches" from occurring. This construction requires more theory to be developed which is detailed in 4.1.4.

3.2 Simulation

Because the target audience of this simuation is an undergraduate studying COD, a simple single stage abbreviated MIPS simulator has been altered to incorporate asynchronous design principles.

3.2.1 The Synchronous Starting Point

The first design is a single state synchronous MIPS simulation capable of using the `addi` (add immediate), `add` (add register), `sw` (save word to memory), `lw` (load word from memory), `j` (jump), and `bne` (branch when not equal) instructions. A simple instruction memory, and data memory have been simulated using behavioral Verilog.[1] An ALU was constructed used structural Verilog code with the naive assumption that all AND, NOT, and OR gates generate one time unit of delay. The adder is a simple cascading design. A simple program has been written to test the functioning of the unit.

```
1  // Instruction Fetch Module
2  // Synchronous Version
3  // Robert Webb
4
5  module instructionfetch(pc, clk, instruction);
6     input [31:0] pc;
7     input         clk;
8     output [31:0] instruction;
9
10    reg [31:0]          instructions[0:56];
11    reg [31:0]          instruction;
12
13    always @ (posedge clk) #10
14      begin
15         if (pc > 60) $stop;
16         instruction = instructions[pc];
17      end
18
19    initial
20      begin
21      instructions[0] = 0;
22      instruction = instructions[0];
```

[1] Behavioral Verilog is a specification of the behavior of a circuit without specifying the physical design of the circuit.

```
23      instructions[4] = 32'b00100000000001010000000000000001; // addi $5, $0, 1
24      instructions[8] = 32'b00100000000001000000000000010100; // addi $4,$0,20
25      instructions[12] = 32'b10101100000000000000000000000000; // sw $0, $0(0)
26      instructions[16] = 32'b00100000000000010000000000000001; //addi $1, $0, 1
27      instructions[20] = 32'b10101100000000010000000000000100; //sw $1, $0(4)
28      //Loop
29      instructions[24] = 32'b10001100000001100000000000000000; //lw $6, $0(0)
30      instructions[28] = 32'b10001100000001110000000000000100; //lw $7, $0(4)
31      instructions[32] = 32'b00000000111001100100000000100000;//add $8, $7, $6
32      instructions[36] = 32'b10101100000001110000000000000000; //sw $7, $0(0)
33      instructions[40] = 32'b10101100000001000000000000000100; //sw $8, $0(4)
34      instructions[44] = 32'b00100000100001001111111111111111; //addi $4, $4, -1
35      instructions[48] = 32'b00010100101001001111111111111001;// bne $4,$5,-7
36      //EndLoop
37      instructions[52] = 32'b00000001000000000001000000100000;// add $2,$8,$0
38      instructions[56] = 32'b00001000000000000000000000001101; // j 52
39      instructions[60] = 32'b0;
40      end // initial begin
41  endmodule // instructionfetch
```

At the conclusion of this program register $2 contains the 20^{th} Fibonacci number 6765. The remainder of the code for the traditional synchronous processor is presented as an appendix.

With the previously stated assumptions, the above program will run with a clock frequency of 96 time units per cycle. At this speed the simulation completes in 13488 time units. These values are noted so they can be compared to the speed of the asynchronous designs.

3.2.2 Beginning the Asynchronous Conversion

Verilog is a modular design tool, so the logical starting point is to define new modules corresponding to dual rail AND, OR, and NOT. Because NOT2 has no logical components, it was assumed to have no delay.

For AND2 and OR2 to work properly, delay must be introduced so the undetermined bits will be logically obvious. Assuming that AND took one tick to transition up and two ticks to transition down while OR instantaneously transitioned up and took one tick to transition down produced the desired behavior. These delays were determined by running the simulation with different delays

while observing the outputs of the ALU. Using these delays, the undetermined bits cascaded down the outputs of the ALU as the carry rippled through the circuit, while guaranteeing that there was always an undetermined bit until all bits were defined. All the busses were then converted to dual rail encoding and all logical gates were converted to dual rail gates.

```
1  // Simple Dual-Rail Gates
2  // Asynchronous Processor
3  // Robert Webb
4
5  module not2(outH, outL, inH, inL);
6     input inH,inL;
7     output outH, outL;
8
9     buf (outH, inL),
10      (outL, inH);
11 endmodule // not2
12
13 module and2(outH, outL, AH, AL, BH, BL);
14    input AH,AL,BH,BL;
15    output outH, outL;
16
17    and #(1,2) (outH, AH, BH);
18    or #(0,1) (outL, AL, BL);
19 endmodule // and2
20
21 module or2(outH, outL, AH, AL, BH, BL);
22    input AH,AL,BH,BL;
23    output outH, outL;
24
25    or #(0,1) (outH, AH, BH);
26    and #(1,2) (outL, AL, BL);
27 endmodule // or2
```

3.2.3 Completion Signals

The only parts of the processor that introduce significant and non-constant delay in this simulation are the 3 ALU units. The first ALU increments the current PC by 4. The second ALU increments the incremented PC by the sign-extended and shifted immediate from the current instruction. The completion signals from these units are combined to generate a signal referred to as `PCready` in the simulation specification. The final ALU performs the calculation required by the instruction. The completion signal from this unit is referred to as `ALUready`

in the simulation specification. To determine if an ALU has completed it's work, a module in Verilog is used. The basic circuit is shown in figure 3.4.

Depending on the situation, all three may need to be finished to move on. If the instruction is not a branch, then the ALU that is computing `branchPC` is computing garbage, so the ALU adding the shifted, sign extended immediate and the incremented PC does not need to be finished. The simulated processor does take advantage of this fact in that the `PCready` signal is computed using the output of the multiplexor that selects `nextPC`. This allows `PCready` to signal without waiting for `branchPC` if it is not needed.

The completion detector module takes a high and low output from an ALU or other processing unit[2] and returns a high bit if it has completed its work and a low bit if it has not. The determination is accomplished by computing the XOR of each high and low bits and then taking the AND of all of these (called `good` in the code). A BUF with a delay of 3 time units is then used to delay the output of the AND (called `really` in the code) which is then compared to the real-time output of the AND gate. The delayed signal is used to attenuate any "glitches" in the processor that might "jump the gun". Attenuation is accomplished because BUF module in Verilog is internally defined to ignore any signals that do not remain high for the duration of the delay of the module. These signals are then passed through a behavioral bit of code that simply changes the possible bit value of x that is produced by Verilog at the beginning of the simulation into a 0. The behavioral code is only used to get the process started at the beginning because none of the processing units will initiate with an undefined completion signal.

```
1  // Completeness Indicator Module
2  // Asynchronous Processor
3  // Robert Webb
4
5  module ready(inH, inL, out);
6     input [31:0] inH, inL;
7     output         out;
```

[2]It is important that the output of a flip-flop never be used to generate a completion signal in the way described because a flip-flop could enter a metastable state. It is not needed here to generate a completion signal from the register file here, so this detail can be omitted, but in the case a completion signal is needed it can be produced by simply delaying the request signal going into the register file by an upper bound of latency of the flip-flops.

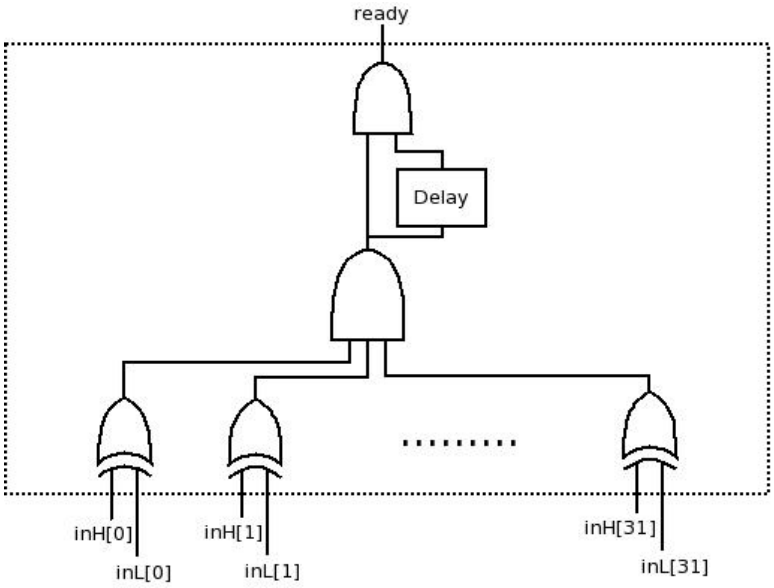

Figure 3.4: Completion Detector

```
8
9       reg                  out;
10      wire [31:0] r;
11      wire good, really, structout;
12
13      xor #1 (r[0], inH[0], inL[0]), (r[1], inH[1], inL[1]),
14        (r[2], inH[2], inL[2]), (r[3], inH[3], inL[3]),
15        (r[4], inH[4], inL[4]), (r[5], inH[5], inL[5]),
16        (r[6], inH[6], inL[6]), (r[7], inH[7], inL[7]),
17        (r[8], inH[8], inL[8]), (r[9], inH[9], inL[9]),
18        (r[10], inH[10], inL[10]), (r[11], inH[11], inL[11]),
19        (r[12], inH[12], inL[12]), (r[13], inH[13], inL[13]),
20        (r[14], inH[14], inL[14]), (r[15], inH[15], inL[15]),
21        (r[16], inH[16], inL[16]), (r[17], inH[17], inL[17]),
22        (r[18], inH[18], inL[18]), (r[19], inH[19], inL[19]),
23        (r[20], inH[20], inL[20]), (r[21], inH[21], inL[21]),
24        (r[22], inH[22], inL[22]), (r[23], inH[23], inL[23]),
25        (r[24], inH[24], inL[24]), (r[25], inH[25], inL[25]),
26        (r[26], inH[26], inL[26]), (r[27], inH[27], inL[27]),
27        (r[28], inH[28], inL[28]), (r[29], inH[29], inL[29]),
28        (r[30], inH[30], inL[30]), (r[31], inH[31], inL[31]);
29      and #2 (good, r[0], r[1], r[2], r[3], r[4], r[5], r[6], r[7], r[8],
30              r[9], r[10], r[11], r[12], r[13], r[14], r[15], r[16], r[17],
31              r[18], r[19], r[20], r[21], r[22], r[23], r[24], r[25], r[26],
32              r[27], r[28], r[29], r[30], r[31]);
33      buf #3 (really, good);
34      and #2 (structout, really, good);
35      always #1
36        if(structout == 1) out = 1; else out = 0;
```

```
37 endmodule // ready
```

3.2.4 Clock Removal

In a single cycle implementation, the clock triggers 4 events:

1. Signal an update of the PC.
2. Signal a new instruction fetch.
3. Signal a write back of data to the register file.
4. Signal a write or read of the memory file.

The goal of the simulation is to somehow generate a "global completion signal" (GCS) which will replace the clock.

First, a distinction between the write or read function of the memory module is required. A read can happen when the ALU is finished because it is calculating the address of the read. This must occur before any GCS because the value of the read may be required by the register file, which will write back upon receiving the GCS. A write would occur upon receiving the GCS. Therefore, the memory module has been written to have distinct `get` and `set` signals. The `get` signal is connected to the `ALUready` signal. The `set` signal is connected to the GCS.

```
1  // Memory Module
2  // Asynchronous Processor
3  // Robert Webb
4
5  module memoryfile(readdataH, readdataL, addressH, addressL,
6                    writedataH, writedataL, memwrite, memread, get, set);
7     input [31:0] addressH, addressL, writedataH, writedataL;
8     input         get, set, memwrite, memread;
9     output [31:0] readdataH, readdataL;
10
11    reg [31:0]          readdataH = 0, readdataL = 0;
12    reg [31:0]          sys[127:0];
13
14    always @ (posedge get)
15      begin
16         if (memread)
```

```
17            begin
18               readdataH = sys[addressH]; readdataL = ~sys[addressH];
19               $strobe("Read %d from ad. %d  %d \n", readdataH, addressH, $time);
20            end
21      end
22    always @ (posedge set)
23      begin
24         if (memwrite)
25           begin
26              sys[addressH] = writedataH;
27              $strobe("Wrote %d to ad. %d  %d \n", writedataH, addressH, $time);
28           end
29      end
30  endmodule // memoryfile
```

3.2.5 Distribution of the GCS

To fully remove the clock, three `ready` modules for the PC, the main ALU, and the final output are used. These ready signals are combined with an AND gate to produce the GCS. The global ready signal is sent back into the PC, instruction fetch, register, and memory modules. Appropriate delays for the GCS going into each module are required to ensure correct operation. Specifically, the ready signal must first perform any writes to the memory or register units before the state changes. The latch holding the current PC can then be updated. The instruction fetch module must wait for a stable output from the latch to lookup the next instruction. A high level schematic of the processor, without control signals, is shown in figure 3.5. The processor is now asynchronous and independent of a clock signal.

3.3 Results and Discussion

3.3.1 Design Lessons

When this design was complete, one major problem was found. Branch instructions always caused the processor to go to a garbage address. The problem

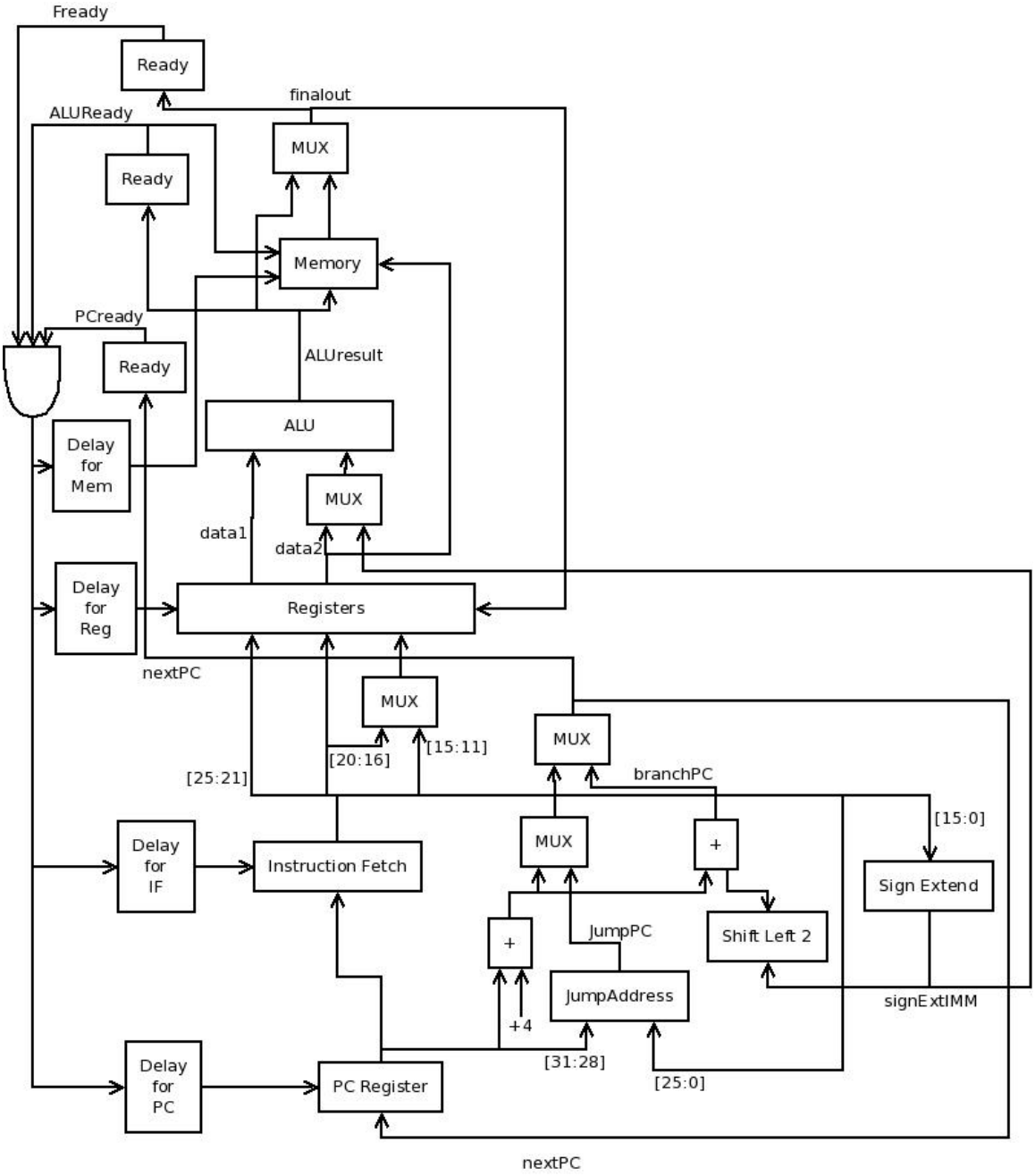

Figure 3.5: The single cycle asynchronous processor excluding control

was the original design simply checked if the main ALU was finished before signaling the next state change. After tracing through the simulation several times, it was determined that the branch address ALU calculation was not completed, so the simulation was jumping to a garbage address. Adding completion signals to all the ALUs fixed the problem, which occurs because it takes much longer for a cascade adder to subtract a small number than to add a small number, and the branch instruction is performing a subtraction to calculate `branchPC`. The bug is instructive of the design issues that asynchronous processor designers must grapple with that may not be as pervasive in a synchronous environment.

3.3.2 Results

The same program that took the synchronous processor 13488 time units to complete was completed by the asynchronous version in 7604 time units. This is a 43.6% increase in computing speed.

A significant amount of the gain is from the simple arithmetic done at the beginning of the Fibonacci calculation. Simple arithmetic strongly favors the asynchronous implementation because it can be completed by the ALU quickly. One would expect the speed gains to be less significant with more complex calculations that approached the worst case delay of a synchronous design.

Chapter 4

An Asynchronous Pipelined MIPS Processor

4.1 Theory

As stated in chapter 2 the theory of asynchronous circuits begins with the "Speed Independent" (SI) model developed in the 50's and 60's by David Muller.[10] This theory assumes that all wires are ideal and have no delays and that gates can have arbitrary and potentially variable delay. A correctly designed SI circuit will never "glitch", meaning that once the inputs are stable, the output bits will make a single transition to the correct output. An example of a "glitch" is described in section 3.1.2. Theoretically correct asynchronous circuits will either produce a valid correct output, or an output that clearly communicates that it is not ready.

4.1.1 Delay Insensitivity

Early computers did not run at a speed where wire delay would be an issue. As silicon design began to increase both clock speed and the number of gates that

could be used, wire delay became a significant part of the delay of the circuit. Because of this development, the logical next step was to develop a theory where wires and gates could have an arbitrary and potentially variable positive delay, known as DI. However, as was shown by Martin in [7], the class of circuits that are DI is very limited, so this theory was largely abandoned. When modern literature refers to DI, generally the assumptions are that of QDI.

4.1.2 Quasi-Delay Insensitivity

The standard compromise to DI is "Quasi-Delay Insensitivity" (QDI). Using the QDI paradigm a designer can assume that a wire that forks will have the same delay in a local area. These are called "iso-chronic forks". In other words, if a wire forks and goes to multiple nearby gates, a transition on that wire will reach the gates at the same time. It has been shown that these circuits are Turing complete.[2]

From an electrical engineering perspective, QDI is a reasonable assumption that can be guaranteed with careful layout of a processor. However, complexity to the design process in increased because the forks that need to be iso-chronic have to be identified and monitored through fabrication to ensure correct operation. In the simulation, the forks required to be iso-chronic are noted.

4.1.3 The Muller C Element

The basic building block of SI and QDI circuits is the Muller C element, first used in the ALU of Iliniac II. As noted in [12], the necessity of a new gate for asynchronous processors can be best described by the concept of "indication". A gate is said to indicate an input if the value of the input can be completely determined by the output of the gate. Consider the two input AND gate. A transition of the output to high indicates that *both* inputs are high. However, a transition to low indicates that only one input is low, so no information can be determined about one of the inputs. Transitioning to high indicates the state of

both inputs, but transitioning to low indicates the state of only one. Similarly the OR gate also does not indicate both inputs on both transitions.

For correct SI or QDI behavior, a gate that indicates both inputs on both transitions is required. It is useful to consider joining two request signals. If both are high, then the output should be high, and if both are low, then the output should be low. If only one is high, then the output should be low. A high joined signal means that that the processing for both is finished. A low combined signal means that both units are ready to do something else.[1] Therefore, if two requests are high and then one goes low, then their joined request should *stay high* because *both* of them are not ready to start processing again. The joined request should only go low when *both* of them are again low. This is the gate known as the Muller C element. It stays low until all inputs are high, then stays high until all inputs are low. The truth table is:

A	B	C
0	0	0
1	0	C_{t-1}
0	1	C_{t-1}
1	1	1

The gate description and transistor implementation are shown in figure 4.1. The code that has been written for 2, 3, and 32 input C elements follows this paragraph. Behavioral Verilog is used because these units are considered atomic. While they could be expressed using AND and OR gates, defining the C element with real AND and OR gates could introduce "glitches" because of different delays in AND and OR gates.

```
1  // C Elements
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module Celement(a, b, c);
6     input a, b;
7     output c;
8
9     reg           c = 0;
10
11    always #1 if(a == 1 && b == 1) c = 1;
```

[1] Request signals will be more completely discussed in section 4.1.6.

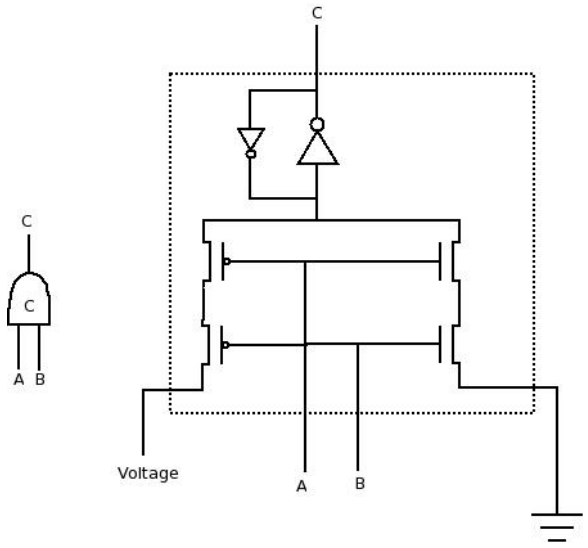

Figure 4.1: C element: gate description and transistor implementation

```
12    always #1 if(a == 0 && b == 0) c = 0;
13 endmodule // Celement
14
15 module Celement3(a, b, c, d);
16    input a, b, c;
17    output d;
18
19    reg           d = 0;
20
21    always #1 if(a == 1 && b == 1 && c == 1) d = 1;
22    always #1 if(a == 0 && b == 0 && c == 0) d = 0;
23 endmodule // Celement3
24
25 module CelementBUS(r, out);
26    input [31:0] r;
27    output         out;
28
29    reg                 out = 0;
30
31    always #1 if( r == 32'b11111111111111111111111111111111 ) out = 1;
32    always #1 if( r == 32'b0 ) out = 0;
33 endmodule // CelementBUS
```

It should be noted that a C element has hysteresis, so is a form of latch. In fact, many asynchronous circuits use C elements as part of their state memory. An example is shown in the dual-rail pipeline registers in figures 4.11 and 4.12.

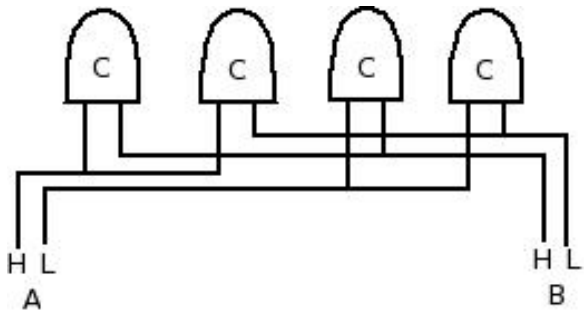

Figure 4.2: C Element minterms for a 2 input DIMS circuit

4.1.4 DIMS

Once C elements have been defined it is possible to make circuits that, by definition, cannot "glitch" using a systematic algorithm. The algorithm is called "Delay Insensitive Minterm Synthesis" or DIMS. Using this method all of the data must be in dual-rail format as defined in section 3.1. The value of $(1, 1)$ is not allowed in this case. The circuits will be designed so that that combination will never be produced as long as that combination is never an input and all transitions on inputs are to or from $(0, 0)$. Specifically, an input of $(1, 0)$ can never transition directly to $(0, 1)$ or vise-versa.

As an example, a simple 2 input circuit will be converted.[2] Because there are 4 possible input states, there will be 4 C elements in the circuit representing each possible minterm.[3] Figure 4.2 represents the first stage of the DIMS circuit.

The truth table for the AND gate is:

A	B	C
0	0	0
1	0	0
0	1	0
1	1	1

To complete the DIMS algorithm the C element outputs associated with each minterm are connected to the appropriate output. If there are multiple minterms for a given output, they can be connected with an OR gate as each minterm is

[2]Note that because these are dual-rail circuits, a 2 input circuit will have 4 input wires.
[3]If not all inputs are possible, then some of the C elements can be removed.

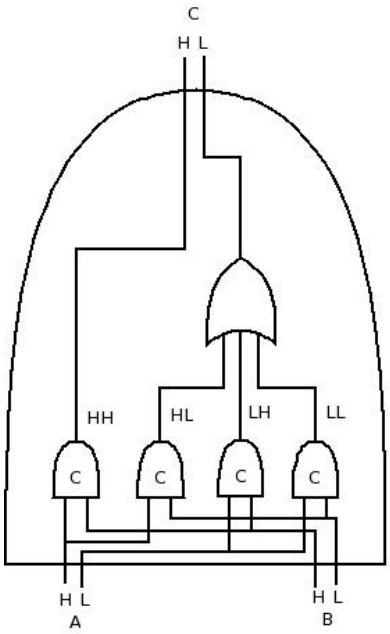

Figure 4.3: DIMS AND gate

exclusive from all the others. The resulting DIMS AND gate is show in figure 4.3. Using this method, it is simple to make a DIMS OR gate, as seen in figure 4.4. The verilog code for these modules also follows.

```
1  // DIMS Asynchronous Gates
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module andD(outH, outL, AH, AL, BH, BL);
6     input AH, AL, BH, BL;
7     output outH, outL;
8
9     wire   LL, LH, HL, HH;
10
11    Celement ll(AL, BL, LL), lh(AL, BH, LH), hl(AH, BL, HL), hh(AH, BH, HH);
12    buf (outH, HH);
13    or #1 (outL, LL, LH, HL);
14 endmodule // andD
15
16 module orD(outH, outL, AH, AL, BH, BL);
17    input AH, AL, BH, BL;
18    output outH, outL;
19
20    wire   LL, LH, HL, HH;
21
22    Celement ll(AL, BL, LL), lh(AL, BH, LH), hl(AH, BL, HL), hh(AH, BH, HH);
```

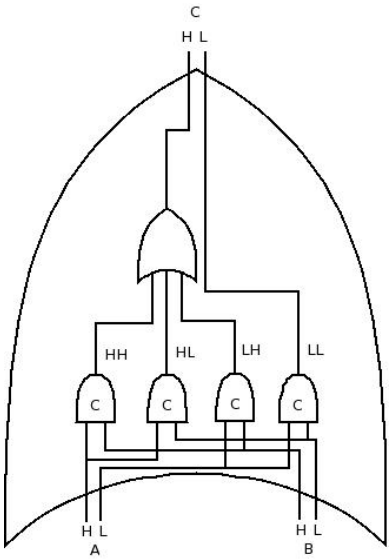

Figure 4.4: DIMS OR gate

```
23      buf (outL, LL);
24      or #1 (outH, HH, LH, HL);
25  endmodule // andD
```

The resulting circuits will satisfy all the requirements of the SI theory and the QDI theory. The only delay assumption is that the forks of the wires going into the C elements are isochronic. However, to use these circuits, the inputs must begin in the undefined state (i.e. all bits are at $(0, 0)$). By definition, in this state, the output will also be in the undefined state. The inputs can then change *monotonically* to their input state. In complex circuits with multiple outputs, the behavior of the outputs can be defined to be strongly indicating or weakly indicating. Strongly indicating circuits require all inputs to be defined before producing any output. Weakly indicating circuit outputs are allowed to become defined at different times and will produce output before all inputs are defined, if possible. For example, in the adder that is described in section 4.2.2, the single bit adders can produce output if the carry input is undefined because the carry out can be determined in some cases without the carry in value.

Once the output is defined, care must be taken to be sure that all inputs return to the undefined state and that the inputs remain undefined until the

output has again returned to the undefined state. This is very important because of the hysteresis of the C elements. One possible consequence of not returning to the undefined state would be an output where both the high bit and low bit of an output is 1, which is not allowed in the circuit specification. The pipeline structure will ensure that the inputs and outputs return to the undefined state because the pipeline timing signals will be used to return the input to the undefined state and ensure the output has returned to the undefined state.

4.1.5 Design Choices

Clearly the new AND and OR gates are much more complicated. A traditional AND gate has 6 transistors and the new one has 30 transistors. For this reason, the approach of making new AND and OR modules and changing all of the gates in a synchronous processor as was done in chapter 3 is unrealistic. Instead of doubling the transistor count as before, there would be a quintupling of the transistor count. To mitigate this, when converting a circuit the DIMS procedure is done on larger modules to reduce replication of overhead. However, this can quickly become daunting. Assuming there are no possible optimizations, an n input DIMS gate will have 2^n C elements each having n inputs!

Modules have been created to convert between dual-rail data and a bundled data encoding. In dual-rail encoding of data, the validity of the data is encoded as part of the data. This works well when the data is being used by combinatorial logic and makes the calculation of a completion signal much easier, but is very cumbersome in other parts of a data path such as multiplexors and registers. For these parts of the data path bundled data is used. Bundled data is the traditional encoding of the data with an additional bit indicating if the data is valid. This additional bit is usually referred to as a "request" because if it is high, then the data is requesting to be consumed by some subsequent processing unit.

The dual-rail to bundled data converter is labeled in schematics as "SR", and the bundled data to dual -rail converter is labeled as "DR". The circuit diagram for the "DR" converter is in figure 4.5. The circuit diagram for the

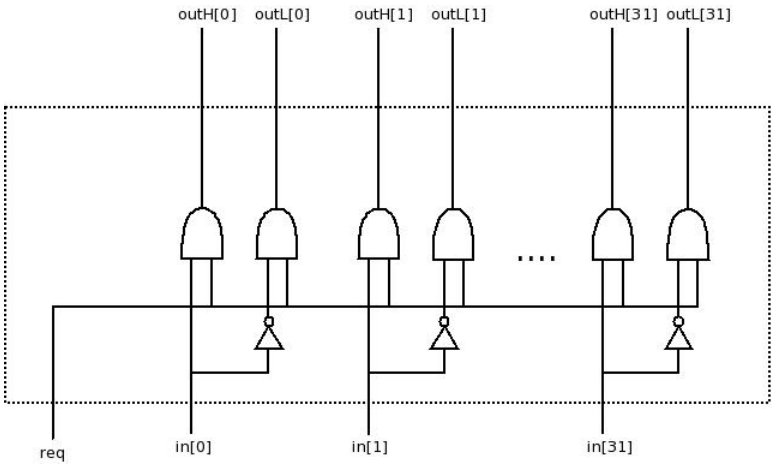

Figure 4.5: Conversion Unit, Bundled Data into Dual-rail

"SR" converter is in figure 4.6. The Verilog code for these modules also follows. The "DR" module is very important because it is how the input for a DIMS unit will return to the undefined state before transitioning to the next set of data inputs. The `req` input to the "DR" cannot go high until the `req` output of the "SR" goes low. Most of these control issues will be handled by the pipeline control units defined in section 4.1.6.

```
1  // Conversion Modules
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module b2dBIT(inp, req, outH, outL);
6     input inp, req;
7     output outH, outL;
8
9     wire   inpL, inpH;
10
11    not #1 (inpL, inp);
12    buf #1 (inpH, inp); // So inpH and inpL transition simultaneously!
13
14    and #1 (outH, inpH, req), (outL, inpL, req);
15 endmodule // bundled2dualBIT
16
17 module bundled2dual(inp, req, outH, outL);
18    input req;
19    input [31:0] inp;
20
21    output [31:0] outH, outL;
22
23    b2dBIT b0(inp[0], req, outH[0], outL[0]), b1(inp[1], req, outH[1], outL[1]),
```

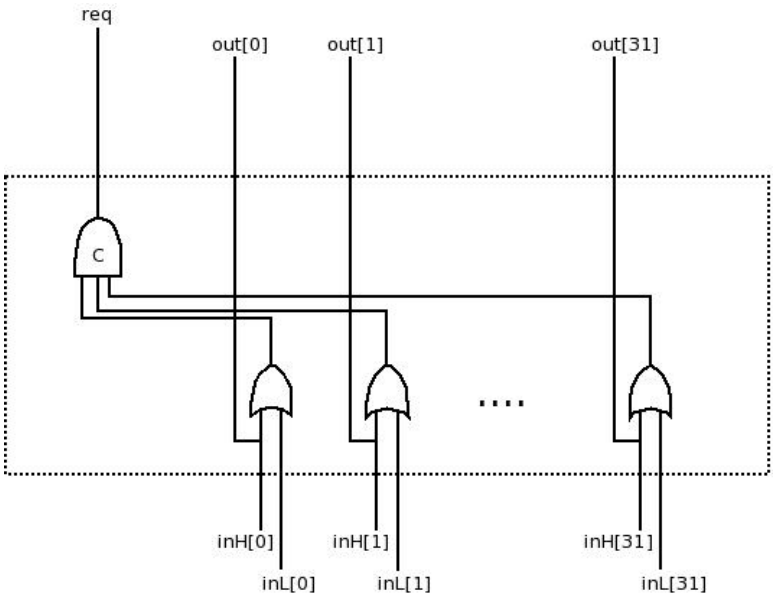

Figure 4.6: Conversion Unit, Dual-rail into Bundled Data

```
24     b2(inp[2], req, outH[2], outL[2]), b3(inp[3], req, outH[3], outL[3]),
25     b4(inp[4], req, outH[4], outL[4]), b5(inp[5], req, outH[5], outL[5]),
26     b6(inp[6], req, outH[6], outL[6]), b7(inp[7], req, outH[7], outL[7]),
27     b8(inp[8], req, outH[8], outL[8]), b9(inp[9], req, outH[9], outL[9]),
28     b10(inp[10], req, outH[10], outL[10]), b11(inp[11], req, outH[11], outL[11]),
29     b12(inp[12], req, outH[12], outL[12]), b13(inp[13], req, outH[13], outL[13]),
30     b14(inp[14], req, outH[14], outL[14]), b15(inp[15], req, outH[15], outL[15]),
31     b16(inp[16], req, outH[16], outL[16]), b17(inp[17], req, outH[17], outL[17]),
32     b18(inp[18], req, outH[18], outL[18]), b19(inp[19], req, outH[19], outL[19]),
33     b20(inp[20], req, outH[20], outL[20]), b21(inp[21], req, outH[21], outL[21]),
34     b22(inp[22], req, outH[22], outL[22]), b23(inp[23], req, outH[23], outL[23]),
35     b24(inp[24], req, outH[24], outL[24]), b25(inp[25], req, outH[25], outL[25]),
36     b26(inp[26], req, outH[26], outL[26]), b27(inp[27], req, outH[27], outL[27]),
37     b28(inp[28], req, outH[28], outL[28]), b29(inp[29], req, outH[29], outL[29]),
38     b30(inp[30], req, outH[30], outL[30]), b31(inp[31], req, outH[31], outL[31]);
39 endmodule // bundled2dual
40
41 module dual2bundled(inpH, inpL, out, req);
42     input [31:0] inpH, inpL;
43
44     output [31:0] out;
45     output          req;
46
47     wire [31:0]          r;
48
49     assign out = inpH;
50
51     or #1 (r[0], inpH[0], inpL[0]), (r[1], inpH[1], inpL[1]),
52       (r[2], inpH[2], inpL[2]), (r[3], inpH[3], inpL[3]),
```

```
53      (r[4], inpH[4], inpL[4]), (r[5], inpH[5], inpL[5]),
54      (r[6], inpH[6], inpL[6]), (r[7], inpH[7], inpL[7]),
55      (r[8], inpH[8], inpL[8]), (r[9], inpH[9], inpL[9]),
56      (r[10], inpH[10], inpL[10]), (r[11], inpH[11], inpL[11]),
57      (r[12], inpH[12], inpL[12]), (r[13], inpH[13], inpL[13]),
58      (r[14], inpH[14], inpL[14]), (r[15], inpH[15], inpL[15]),
59      (r[16], inpH[16], inpL[16]), (r[17], inpH[17], inpL[17]),
60      (r[18], inpH[18], inpL[18]), (r[19], inpH[19], inpL[19]),
61      (r[20], inpH[20], inpL[20]), (r[21], inpH[21], inpL[21]),
62      (r[22], inpH[22], inpL[22]), (r[23], inpH[23], inpL[23]),
63      (r[24], inpH[24], inpL[24]), (r[25], inpH[25], inpL[25]),
64      (r[26], inpH[26], inpL[26]), (r[27], inpH[27], inpL[27]),
65      (r[28], inpH[28], inpL[28]), (r[29], inpH[29], inpL[29]),
66      (r[30], inpH[30], inpL[30]), (r[31], inpH[31], inpL[31]);
67    CelementBUS dataCHECK(r, req);
68  endmodule // dual2bundled
```

4.1.6 Muller Pipeline

To understand the asynchronous meaning of a pipeline, it is useful to think about a line of people passing buckets of water to a fire crew. If this line were a synchronous pipeline, everyone would be passing buckets at exactly the same frequency and buckets full of water would appear at the beginning of the pipeline at that exact frequency. However, in an asynchronous pipeline, everyone on the line would be working at their own pace, and each individual pace could change over time. If one person was slow, then everyone before that person in the line would have to wait to pass the next bucket. If a group of people at the end of the line were all fast, then when a bucket got to that part of the line it would speed through, and those people would be waiting idle for the next bucket.

What is required is a handshaking protocol in the form of requests and acknowledgments. The solution, called the four-phase handshake protocol and generally credited to Muller, focuses on the actions of an individual unit of the circuit.[12] Each unit has four timing signals, two requests for receiving and passing along and two acknowledges, for receiving and passing along. The wave forms of corresponding request and acknowledge signals are shown in figure 4.7.

A unit begins at an idle state where all timing signals are low. For simplicity, the unit will be referred to as current unit (CU). The previous unit (PU) and next

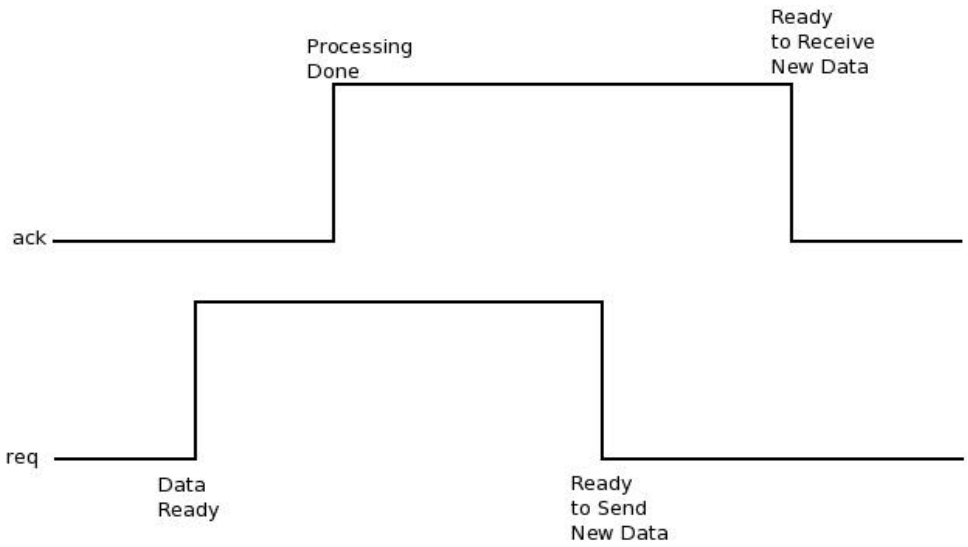

Figure 4.7: The 4 phase handshake protocol waveform

unit (NU) will also be referenced. All signals are currently low. At this point, CU is ready to receive a request from PU. When the request from PU is received, CU does its work, then sends a request ahead to NU. CU now must wait. At this point, CU is still receiving a request from PU, because no acknowledgement has been send back to PU. Also, CU has to continue to output the current work until NU acknowledges receipt. When CU gets acknowledgement from NU, CU can send an acknowledgement back to PU. When CU sends the acknowledge back, it is not to say "I have it", it is to say "I have successfully passed it on".[4] When the acknowledge goes high, that *does not* communicate that NU is ready for new data. Recall back to section 4.1.4, once a DIMS circuit is finished, it must go back to the neutral state. CU can't go back to the neutral state right now, because the request from PU is still high and the acknowledge from NU is still high. However, PU will lower its request once it receives acknowledgement. When PU lowers its request, the inputs to CU will be in the neutral state. The neutral inputs will propagate through the logic in CU, and when the outputs are also neutral, the request to NU will go low. Once NU has returned to the neutral state, it will set

[4]This is important because CU needs to have valid inputs, which are signaled by the request from the previous unit, and when CU produces output, that output must be put somewhere (usually some sort of pipeline register) *before* the inputs change.

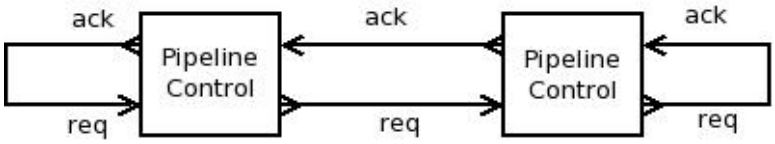

Figure 4.8: A basic asynchronous pipeline made from pipeline control units

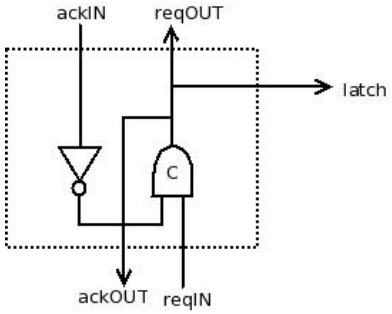

Figure 4.9: Muller Pipeline Unit

acknowledge low. CU can then set acknowledge to PU low. CU is now ready to get a new request, as all the signals have returned to 0.

To simplify the details described above, a pipeline control unit is needed. This unit will deal with most of the handshaking independent of the actual processing units. The pipeline control unit can be copied and placed in sequence to coordinate the handshaking actions. The processing units would then only be required to receive a request and generate a request.

The use of this simple pipeline control unit is show in figure 4.8. Muller designed a circuit that would produce this behavior, as shown in figure 4.9. In addition to generating the request and acknowledge signals, this gate also generates a latch enable, so that data can be carried down the pipe with the signals as shown in figure 4.10. An alternative pipeline control unit can be used with dual-rail data as shown in figure 4.11. The C elements in this pipeline control unit are used instead of a traditional latch. This idea can be extended to multiple bits of dual-rail data as shown in 4.12.

As show in figure 4.10, all the pipeline does is propagate data down each of the stages. To make this useful processing units need to be added. For the

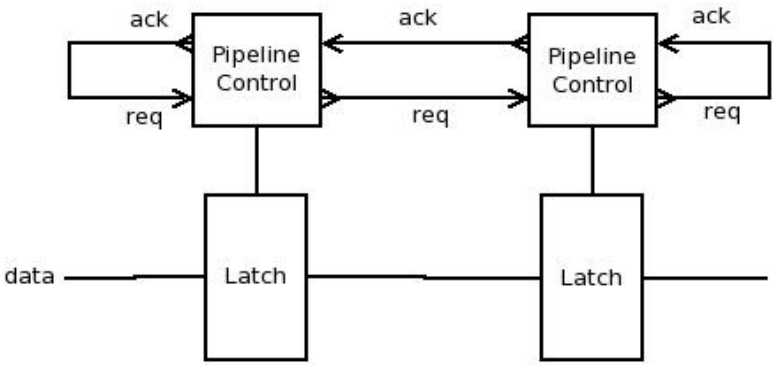

Figure 4.10: A basic asynchronous pipeline with latches

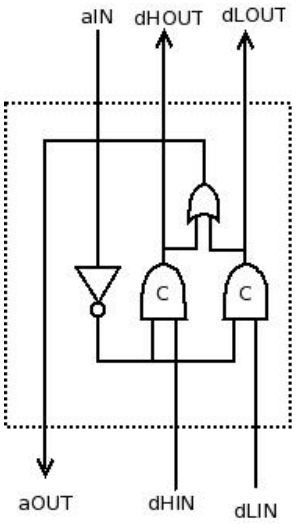

Figure 4.11: Dual-Rail Pipeline Unit

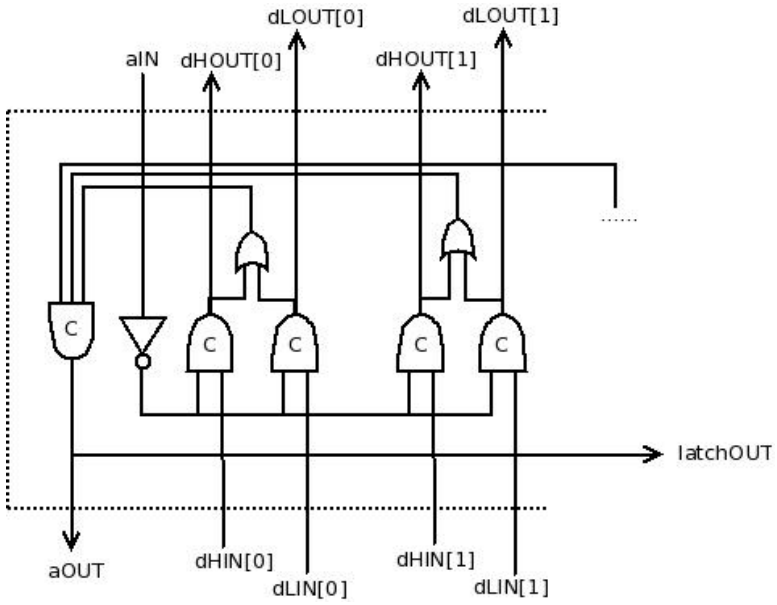

Figure 4.12: Dual-Rail Pipeline Gate with Multiple data Inputs

purposes of this paper, there are two kinds of processing units: constant delay units and variable delay units. Inserting a constant delay unit like a register file is shown in figure 4.13. The only variable delay unit in this project is the ALU. Inserting one of those is a bit more complex because the data is in bundled form when traveling down the pipeline, so the converters defined in figures 4.5 and 4.6 must be used. The result is shown in figure 4.14.

For the ALU to function correctly the inputs must return to the neutral state before the next set of values become defined. Initially all the values of the timing signals are 0 which causes the DR module to only output the neutral state. First `reqIN` = 1 which allows the ALU to begin work by defining the inputs. Then `reqOUT` = 1 when it is finished because the SR module generates a completion signal. Then `ackIN` = 1 when the next phase has latched the data, and `ackOUT` = 1 because it is directly connected to `ackIN`. Because of the acknowledge, `reqIN` = 0 when the previous unit lowers its request and the ALU inputs go to neutral because of the DR unit. After processing, the ALU outputs go neutral and SR unit causes `reqOUT` = 0. This allows `ack` = 0, and the initial state has been restored.

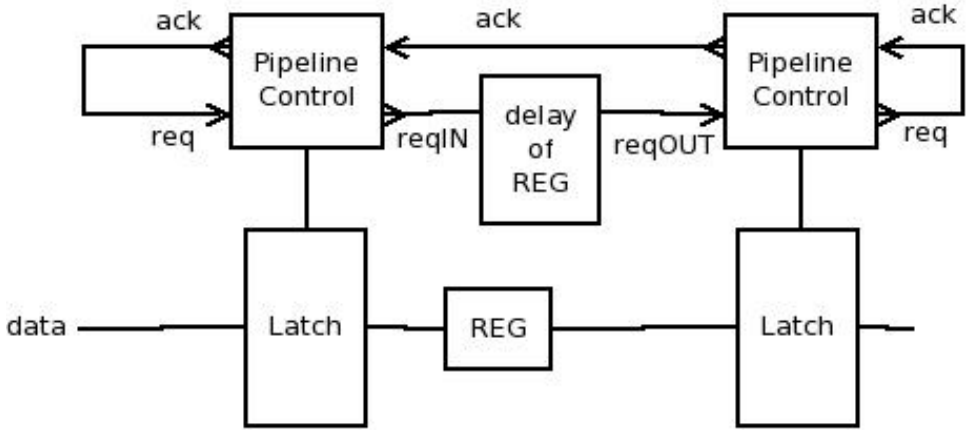

Figure 4.13: A basic asynchronous pipeline with a constant delay processing unit

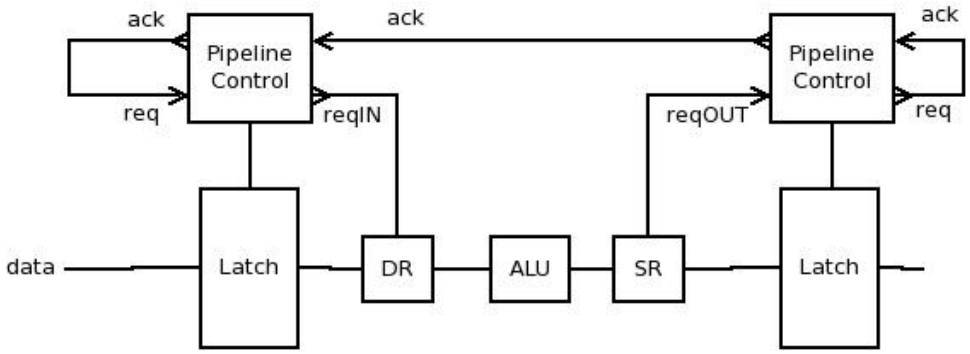

Figure 4.14: A basic asynchronous pipeline with a variable delay processing unit

4.2 Simulation

With section 4.1 completed, the necessary theory to complete an asynchronous MIPS pipeline is available.

4.2.1 Data Path Decomposition

Asynchronous MIPS data paths are generally decomposed into 3 stages: Instruction Fetch, Instruction Decode, and Execute. The memory and write-back stages of a traditional five stage MIPS pipeline are incorporated into the execute stage because many of those operations can occur in parallel if some circuitry is duplicated. This parallel execution can be exploited to a great degree in asynchronous processors. For a more detailed discussion of this point, see [3, 8].

Instruction Fetch

The Instruction Fetch (IF) stage is pictured in figure 4.15. Because this is the first stage, it must generate the first request. This is done by combining the request from the two main units of this stage in a C element. The IF unit, because it is a memory unit, will generate its own request when memory reads have been completed. The code that defines the behavior of the IF unit is in appendix C. The adder that calculates the PC incremented by 4 will also generate a completion signal when it has finished its work. It should be noted that since there is no request coming into the instruction fetch stage, the validity of the inputs to the adder, which is a DIMS implementation, is determined by the opposite of the acknowledge from the Instruction Decode (ID) stage.

Jumps are handled entirely in this stage by simply cycling through instructions and the PC that is determined by the jump instructions until a non-jump instruction is found. This process is entirely transparent to the subsequent stages. It appears to them as though jump instructions are simply never encountered. This also means that a branch-delay-slot after a jump is unnecessary.

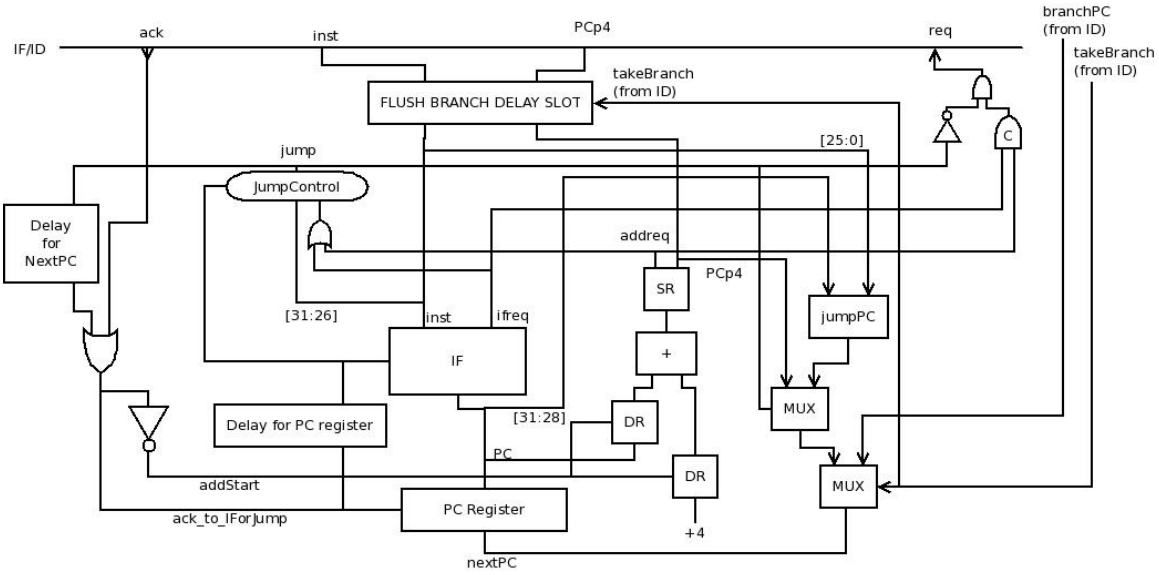

Figure 4.15: Instruction Fetch Stage with control for jump commands

The IF stage attempts to fetch and pass the instruction following a branch regardless of the branch outcome. However, because the IF stage cannot move forward without the acknowledgement of the ID stage, the `branchPC` and `takeBranch` flag will be valid before the `nextPC` is written into the PC register, and the `takeBranch` flag can flush the branch delay slot instruction before the values are saved by the pipeline registers.

Instruction Decode

The ID stage is pictured in figure 4.16. The control signals are generated, and the reads from the register file are completed. The request going forward to the Execute (EX) stage is generated by the constant delay of the multiplexors and latches in the register file and the request from the adder that is calculating the `branchPC`. As the adder is frequently adding a negative number, the delay for this branch becomes significant because all the bits generally have to wait for a long ripple carry. To deal with this problem, if the `branchPC` is not needed, the unit does not wait for the request from the adder.

The positive edge of acknowledge from EX is used to signal a write back into

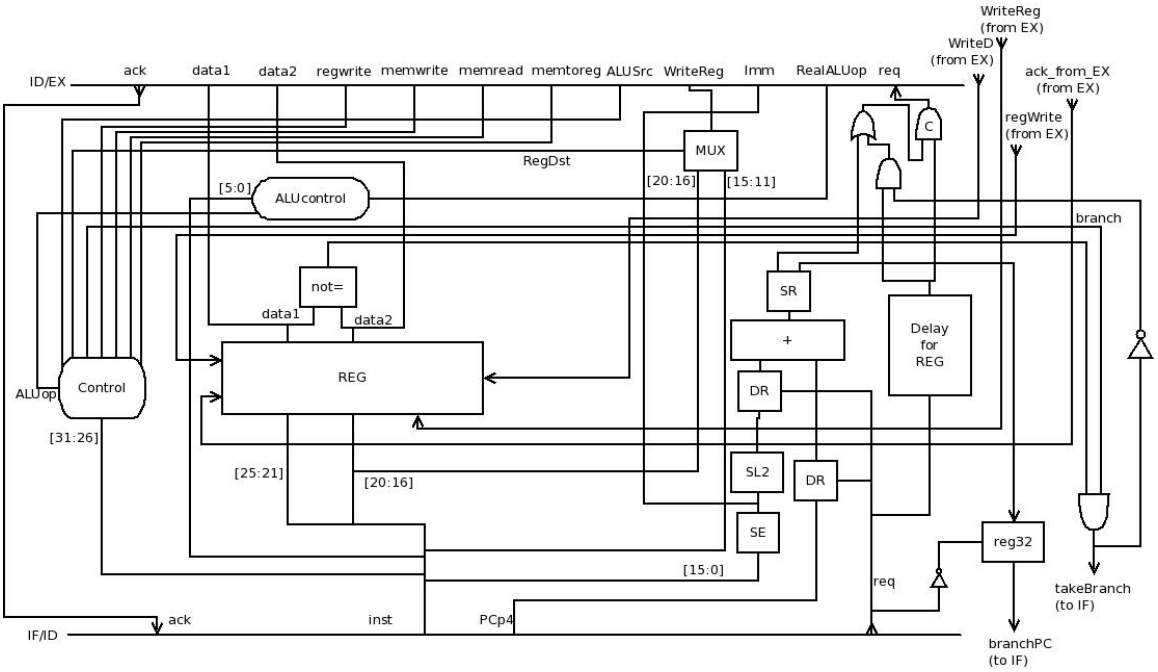

Figure 4.16: Instruction Decode Stage with control and branch decision

the register file. This means that there is no need for a forwarding unit as the pipeline register latches the data coming from the register file on the following negative edge of the acknowledge, which is more than enough time to ensure that the data has been written through.

A latch for the `branchPC` that is triggered by the negative edge of the request into ID is necessary because when the request goes low, the values going into the adder return to the neutral state and eventually `branchPC` also returns to the neutral state, which looks like a zero after it has gone through the conversion to bundled data unit. Because the `branchPC` is not used until the acknowledge to IF is set, its value needs to be saved.[5]

Execute

The EX stage is pictured in figure 4.17. There are a few things worth noting. First there is a delay on the request so the multiplexor that chooses the second

[5]This was a terrible bug to try to find. One doesn't usually think about outputs of an ALU going to 0 between every transition. Every branch was a jump to 0 for much to long...

operand for the ALU is completed before the operands are converted into dual-rail encoding. This is *very* important because the inputs must make a single change from neutral to their valid state. If the multiplexor is not stable the output of the ALU could be invalidated. The ALU produces a request which is passed through to the memory module. The memory is external and generates its own completion signal. The code defining the behavior of the memory module is in appendix C. In the case that memory is not used, the ALU request is simply passed through. In the case of a write the request is passed through and the write is done asynchronously. In the case of a read, then additional delay is produced while the read takes place. This delay could be highly variable because of the different levels of memory cache. The signal is then delayed because the multiplexor that chooses the write back value must be completed before the acknowledge is given.

Because of the intended audience of the simulation, there is not much happening in this stage. However, most of the performance improvement can come from here. In the Cal Tech designs this stage was more of a macro stage with *many* finely pipelined and mostly concurrent processing units. For example, the memory module had its own adder for address calculation. This allowed a read instruction to be dispatched to the memory module, and if there was a cache miss and long delay, the IF and ID stages could continue to dispatch independent instructions to several executions units in parallel. Because there were multiple execution units, there were several busses to write back to the register file. A forwarding unit was required in this case because the writes to the register file happened asynchronously and there was no guarantee that a write would happen before it was needed for the an instruction being dispatched to a different execution unit. Even given all the complexity, the forwarding unit is still more simple than the one required for the 5 stage asynchronous MIPS pipeline.

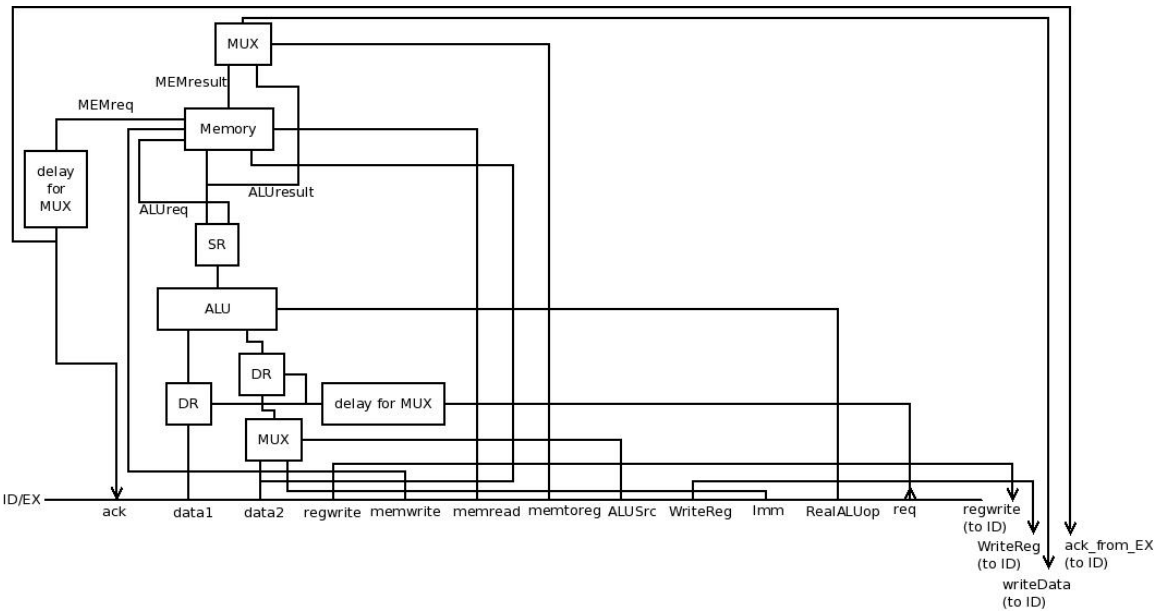

Figure 4.17: Execute Stage with wired for Memory Bypass

4.2.2 ALU

The basis of the ALU is a DIMS single bit adder module that is then used to create a ripple carry adder and ultimately a complete ALU. The truth table for the single bit adder is:

cin	A	B	result	cout
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

As there are 8 possible inputs there will be 8, 3 input, C elements. The values for `result` have been generated in the usual way by combining together the minterms. However, there is an optimization for the `cout` values. If the `cout` values were determined in the usual way, then every operation done by an adder would require a cascade through all the bits because the `cout` of bit n is the `cin` of bit $n+1$. This can be avoided by noticing that if `a` and `b` are both 0, then the

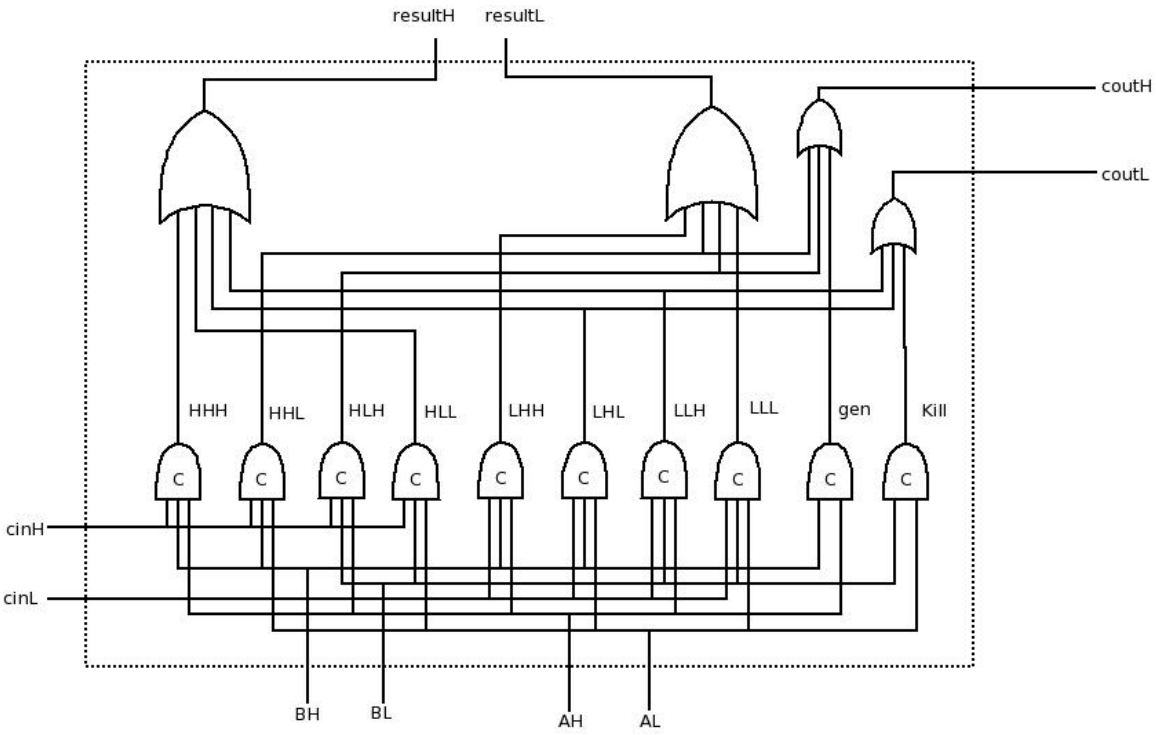

Figure 4.18: DIMS Adder with Generate and Kill carry

`cout` must be 0 so the adder has "killed the carry". Similarly, if both are 1, the `cout` must be 1 so the adder has "generated the carry". Creating these signals and combining them into the DIMS specification produces the circuit shown in figure 4.18. The source code for the adder module follows.

```
1  // DIMS 1-bit Adder
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module adder1(aH, aL, bH, bL, cinH, cinL, coutH, coutL, resultH, resultL);
6     input aH, aL, bH, bL, cinH, cinL;
7     output resultH, resultL, coutH, coutL;
8
9     wire   LLL, LLH, LHL, LHH, HLL, HLH, HHL, HHH, gen, kill;
10
11    Celement3 lll(aL, bL, cinL, LLL);
12    Celement3 llh(aL, bL, cinH, LLH);
13    Celement3 lhl(aL, bH, cinL, LHL);
14    Celement3 lhh(aL, bH, cinH, LHH);
15    Celement3 hll(aH, bL, cinL, HLL);
16    Celement3 hlh(aH, bL, cinH, HLH);
17    Celement3 hhl(aH, bH, cinL, HHL);
18    Celement3 hhh(aH, bH, cinH, HHH);
```

```
19      Celement GEN(aH, bH, gen);
20      Celement KILL(aL, bL, kill);
21
22      or #1 (resultH, LLH, LHL, HLL, HHH),
23        (resultL, LLL, LHH, HLH, HHL),
24        (coutH, LHH, HLH, gen),
25        (coutL, LHL, HLL, kill);
26  endmodule // adder1
```

The ALU unit has several multiplexors to decide between the possible outputs of `and, or, add, sub, slt`. Because DIMS requires that all parts conform to the specification, behavioral code was used to make multiplexors that will return to the neutral state and have monotonic transitions. Structural code could be used to make these modules, but the selector for the multiplexor would have to be converted to dual-rail encoding as well, and the DIMS specification for a 4 input multiplexor would have $2^5 = 32$ C elements, each with 5 inputs.

4.3 Results

The pipelined asynchronous processor finished the program in 8335 time units. This was slightly slower than that single cycle processor described in chapter 3, but that is not entirely unexpected. To create the asynchronous pipeline, a lot of overhead was created. The transistor count has greatly increased. Also, the processor in chapter 3 had very unrealistic timing assumptions with respect to IF and memory access. In the single cycle processor these things happened almost instantly, and the pipeline structure does not allow the information from memory accesses and IF accesses to be used instantly even if they are available. This should not be viewed as a bad thing because in exchange the processor is much more robust.

A possible way to speed up the circuit would be to use the two phase protocol introduced by Ivan Sutherland in his 1988 Turing Award Lecture.[13] In this protocol the request or acknowledge is communicated by a transition up *or* down on the channel. While this could be faster in certain circumstances because it negates the necessity of waiting for all the channels to return to 0, it signifi-

cantly increases circuit complexity, so was not used in this project because of the intended audience.

On a more theoretical note, one primary purpose of pipelining in a synchronous processor is to distribute the tasks that have a fixed delay so that they can be performed in advance while another instruction is performing an operation with variable delay. In a simple synchronous processor, the worst case of that variable delay usually dwarfs the fixed delays, and the clock must run at the worst case delay. In an asynchronous processor no units with variable delay run at their worst case delay except in very unusual circumstances. This would certainly lead to a diminished return for pipelining in an asynchronous setting.

Another factor that might explain the lack of speed increase is the test program. The loop in the test program is short, and the branch is taken almost every time. This would cause the branch delay slot to flush, losing potential instruction time. When combined with the fact that the pipeline has only 3 stages instead of the usual 5, and the loss of a slot every 7 instructions adds up over 20 iterations of a loop.

Chapter 5

Educational Resources

The primary purpose of this project was to generate simulations of asynchronous processors for student use. An example of a synchronous single cycle simulation is presented as the starting point for this work in Appendix A. The basic single cycle asynchronous simulation code is presented in Appendix B. Finally the pipelined asynchronous simulation code is presented in Appendix C.

5.1 Laboratory Exercises

Descriptions of a series of laboratory exercises are included as appendix D. The modular nature of the simulation and the parallel with COD provides straightforward integration into a traditional course straightforward. The six lab exercises vary in difficulty and are also correlated with COD, so an instructor could choose to use as much or as little as desired. If an instructor wanted to add a single lecture and lab to expose students to the concepts of asynchronous design, an overview of the single cycle asynchronous processor in chapter 3 could be presented in lecture. The first suggested lab assignment does not involve any code, only manipulation of constants in existing code to demonstrate understanding of the delay models.

If the instructor wanted to cover asynchronous logic design, another lecture and lab would be the only requirement. The DIMS procedure could be introduced in an hour lecture, assuming a solid understanding of traditional logic design. The second lab described in appendix D, a structural description of the control unit, is already in the curriculum of most architecture courses, and is assumed here. Applying the DIMS procedure as described in the third lab would require an additional $2^6 = 64$ lines of code to make the C elements. These first 64 lines of code would be almost identical. Most students will write the first one, make 63 copies, and modify the copies as required. Some particularly astute students may notice that the module is undefined for most of the 64 possible inputs, so many of those lines are not necessary. To complete the lab, 8 lines of code are necessary to connect the required outputs to the corresponding minterms, so the lab would have an upper bound of approximately 72 lines of code, depending on the optimizations that the students discover in the processor. If more practice with this procedure is required, labs 4 and 5 also convert various parts of the pipelined asynchronous processor using the DIMS procedure. These modules are less complicated than the control unit, so they would also have less than 72 lines of code. For those who want to cover pipelining, combining labs 3, 4, and 5 into a single assignment because of their common theme is advised. A combination of these labs would produce approximately 150 total lines of code, with an optimal solution requiring much less.

For an instructor to introduce the asynchronous pipeline design, an additional 2-3 lectures and 1-2 lab periods would be necessary. Assuming knowledge of the synchronous MIPS pipeline, the lectures would focus on the handshaking protocol, asynchronous pipeline control units, and integrating the constant delay and variable delay processing units into the pipeline. Lab 6 would then involve integrating the product of labs 3, 4, and 5 into the design. There would be very few lines of code required to accomplish this task, a good student will probably find a solution by adding fewer than 50 lines of code to the simulation, but a solid understanding of how the pipeline control and handshaking protocol function would be essential.

5.2 Notes on the use of these simulations

The code for the simulations has been written in structural Verilog wherever possible to ensure the most accurate module of delay. Depending on the level of the class, some modules might be better understood if converted to a behavioral specification. For example, the register file unit is completely specified using multiplexors and flip-flops, which would be beyond the level of most beginning computer architecture students outside of an electrical engineering program.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

This paper demonstrates that asynchronous techniques can be implemented using tools and techniques common in beginning computer architecture courses. While not trivial by any means, the ideas presented in this report have a certain spark that could lead to many teachable moments. Additionally, grasping asynchronous concepts could greatly add to students overall understanding of the synchronous ideas in COD. The traditional architecture instructor could integrate these ideas into a class by adding as little as a single lecture and experimental lab with no code. To go into great depth, adding 4-6 lectures and 3-4 labs to the curriculum would integrate the breath of topics presented in this paper into a traditional course while adding as little as 150-200 lines of code to the workload of a capable student. Because many architecture students, over the course of a term, produce a processor simulation similar to the one presented in appendix A and containing 778 lines of code, this would be an increase of approximately 20-25%.

6.2 Future Work

This project provides substantial avenues for future work.

- Decompose the EX stage into multiple, concurrent micro execution stages.
- Dynamic Scheduling to take advantage of concurrent micro execution stages
- Set up the register file for multiple asynchronous write back values
- Implement forwarding if other modifications make it necessary.
- Implement memory hierarchy and waiting for cache misses.
- Replace behavioral code with structural code.
- Implement more of the MIPS instruction set
- Multiprocessor applications
- Modeling Power consumption differences

Bibliography

[1] A. Davis and S. Nowick, *An Introduction to Asynchronous Circuit Design*, University of Utah, Salt Lake City, 1997.

[2] R. Manohar, A. J. Martin, *Quasi-delay-insensitive circuits are Turing-complete*, Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1995.

[3] A. J. Martin, et al, *The Design of an Asynchronous Microprocessor*, Proc. Decennial Caltech Conference on VSLI, Pasadena, 1989.

[4] A. J. Martin, et al, *The First Asynchronous Microprocessor: The Test Results*, ACM SIGARCH Computer Architecture News, 17(4): 95-98, June 1989.

[5] A. J. Martin, *Tomorrow's Digital Hardware will be Asynchronous and Verified*, Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - information Processing '92, 1(1): 684-695, 1992.

[6] A. J. Martin, *Asynchronous Datapaths and the Design of an Asynchronous Adder*, Cal Formal Methods in System Design: 119-137, 1991.

[7] A. J. Martin, *The Limitations to Delay-Insensitivity in Asynchronous Circuits*, Proceedings of the 6th MIT Conference on Advanced Research in VSLI, MIT Press, Cambridge, 1990.

[8] A. J. Martin, et al, *The Design of an Asynchronous MIPS R3000 Microprocessor*, Proceedings of the Seventeenth Conference on Advanced Research in VLSI, 1997.

[9] A. J. Martin, et al, *Speed and Energy Performance of an Asynchronous MIPS R3000 Microprocessor*, Technical Report, Cal Tech Computer Science Department, Pasadena, 2001.

[10] D. E. Muller and W. S. Bartky, *A Theory of Asynchronous Circuits*, Proc. Int'l Symp. Theory of Switching, Part 1, Harvard Univ. Press, 1959, pp. 204243.

[11] D. A. Patterson, J. L. Hennessy, *Computer Organization & Design: The Hardware-Software Interface, Second Edition*, Morgan Kaufmann, San Francisco, 1998.

[12] J. Sparsø, *Asynchronous Circuit Design: A Tutorial*, Technical University of Denmark, 2006.

[13] I. E. Sutherland, *Micropipelines*, Communications of the ACM, 32(6): 720 738, June 1989.

[14] J. M. Yarbrough, *Digital Logic: Applications and Design*, PWS Publishing, Boston, 1997.

Appendix A

The Complete Synchronous Single-Cycle Source Code

```
1  // Buffer Modules
2  // Synchronous Version
3  // Robert Webb
4
5  module enable(i, rst, enbit);
6     input [4:0] i;
7     input       rst;
8     output [31:0] enbit;
9
10    wire [4:0] ni;
11
12    not (ni[0], i[0]),
13      (ni[1], i[1]),
14      (ni[2], i[2]),
15      (ni[3], i[3]),
16      (ni[4], i[4]);
17
18    and #1 (enbit[0], ni[4], ni[3], ni[2], ni[1], ni[0], rst),
19      (enbit[1], ni[4], ni[3], ni[2], ni[1], i[0], rst),
20      (enbit[2], ni[4], ni[3], ni[2], i[1], ni[0], rst),
21      (enbit[3], ni[4], ni[3], ni[2], i[1], i[0], rst),
22      (enbit[4], ni[4], ni[3], i[2], ni[1], ni[0], rst),
23      (enbit[5], ni[4], ni[3], i[2], ni[1], i[0], rst),
24      (enbit[6], ni[4], ni[3], i[2], i[1], ni[0], rst),
25      (enbit[7], ni[4], ni[3], i[2], i[1], i[0], rst),
26      (enbit[8], ni[4], i[3], ni[2], ni[1], ni[0], rst),
27      (enbit[9], ni[4], i[3], ni[2], ni[1], i[0], rst),
28      (enbit[10], ni[4], i[3], ni[2], i[1], ni[0], rst),
```

```
29      (enbit[11], ni[4], i[3], ni[2], i[1], i[0], rst),
30      (enbit[12], ni[4], i[3], i[2], ni[1], ni[0], rst),
31      (enbit[13], ni[4], i[3], i[2], ni[1], i[0], rst),
32      (enbit[14], ni[4], i[3], i[2], i[1], ni[0], rst),
33      (enbit[15], ni[4], i[3], i[2], i[1], i[0], rst),
34      (enbit[16], i[4], ni[3], ni[2], ni[1], ni[0], rst),
35      (enbit[17], i[4], ni[3], ni[2], ni[1], i[0], rst),
36      (enbit[18], i[4], ni[3], ni[2], i[1], ni[0], rst),
37      (enbit[19], i[4], ni[3], ni[2], i[1], i[0], rst),
38      (enbit[20], i[4], ni[3], i[2], ni[1], ni[0], rst),
39      (enbit[21], i[4], ni[3], i[2], ni[1], i[0], rst),
40      (enbit[22], i[4], ni[3], i[2], i[1], ni[0], rst),
41      (enbit[23], i[4], ni[3], i[2], i[1], i[0], rst),
42      (enbit[24], i[4], i[3], ni[2], ni[1], ni[0], rst),
43      (enbit[25], i[4], i[3], ni[2], ni[1], i[0], rst),
44      (enbit[26], i[4], i[3], ni[2], i[1], ni[0], rst),
45      (enbit[27], i[4], i[3], ni[2], i[1], i[0], rst),
46      (enbit[28], i[4], i[3], i[2], ni[1], ni[0], rst),
47      (enbit[29], i[4], i[3], i[2], ni[1], i[0], rst),
48      (enbit[30], i[4], i[3], i[2], i[1], ni[0], rst),
49      (enbit[31], i[4], i[3], i[2], i[1], i[0], rst);
50  endmodule // enable
51
52  module signextend16to32(imm, out);
53     input [15:0] imm;
54
55     output [31:0] out;
56
57     buf
58       (out[0], imm[0]), (out[1], imm[1]), (out[2], imm[2]), (out[3], imm[3]),
59       (out[4], imm[4]), (out[5], imm[5]), (out[6], imm[6]), (out[7], imm[7]),
60       (out[8], imm[8]), (out[9], imm[9]), (out[10], imm[10]), (out[11], imm[11]),
61       (out[12], imm[12]), (out[13], imm[13]), (out[14], imm[14]),
62       (out[15], imm[15]), (out[16], imm[15]), (out[17], imm[15]),
63       (out[18], imm[15]), (out[19], imm[15]), (out[20], imm[15]),
64       (out[21], imm[15]), (out[22], imm[15]), (out[23], imm[15]),
65       (out[24], imm[15]), (out[25], imm[15]), (out[26], imm[15]),
66       (out[27], imm[15]), (out[28], imm[15]), (out[29], imm[15]),
67       (out[30], imm[15]), (out[31], imm[15]);
68  endmodule // signextend16to32
69
70  module shiftleft2(inp, out);
71     input [31:0] inp;
72
73     output [31:0] out;
74
75     assign out[0] = 1'b0;
76     assign out[1] = 1'b0;
77
78     buf
79       (out[2], inp[0]), (out[3], inp[1]), (out[4], inp[2]), (out[5], inp[3]),
80       (out[6], inp[4]), (out[7], inp[5]), (out[8], inp[6]), (out[9], inp[7]),
```

```
      (out[10], inp[8]), (out[11], inp[9]),  (out[12], inp[10]),
      (out[13], inp[11]), (out[14], inp[12]), (out[15], inp[13]),
      (out[16], inp[14]), (out[17], inp[15]), (out[18], inp[16]),
      (out[19], inp[17]), (out[20], inp[18]), (out[21], inp[19]),
      (out[22], inp[20]), (out[23], inp[21]), (out[24], inp[22]),
      (out[25], inp[23]), (out[26], inp[24]), (out[27], inp[25]),
      (out[28], inp[26]), (out[29], inp[27]), (out[30], inp[28]),
      (out[31], inp[29]);
endmodule // shiftleft2

module jumpPC(top, inp, out);

   input [3:0] top;
   input [25:0] inp;

   output [31:0] out;

   assign out[0] = 1'b0;
   assign out[1] = 1'b0;

   buf
     (out[2], inp[0]), (out[3], inp[1]), (out[4], inp[2]),  (out[5], inp[3]),
     (out[6], inp[4]), (out[7], inp[5]), (out[8], inp[6]),  (out[9], inp[7]),
     (out[10], inp[8]), (out[11], inp[9]), (out[12], inp[10]),
     (out[13], inp[11]), (out[14], inp[12]), (out[15], inp[13]),
     (out[16], inp[14]), (out[17], inp[15]), (out[18], inp[16]),
     (out[19], inp[17]), (out[20], inp[18]), (out[21], inp[19]),
     (out[22], inp[20]), (out[23], inp[21]), (out[24], inp[22]),
     (out[25], inp[23]), (out[26], inp[24]), (out[27], inp[25]),
     (out[28], top[0]), (out[29], top[1]),
     (out[30], top[2]), (out[31], top[3]);
endmodule // jumpPC


// Mux Modules
// Synchronous Version
// Robert Webb

module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;

   wire   s0,w0,w1;

   not #1 (s0, select);
   and #1 (w0, s0, in0),
          (w1, select, in1);
   or #1 (out, w0, w1);
endmodule // mux2

module mux2x5 (in0, in1, select, out);
   input[4:0] in0,in1;
   input      select;
```

```
20     output [4:0] out;
21
22     wire         s0;
23     wire [4:0]         w0,w1;
24
25         not #1 (s0, select);
26         and #1 (w0[0], s0, in0[0]),  (w0[1], s0, in0[1]),
27           (w0[2], s0, in0[2]), (w0[3], s0, in0[3]),
28           (w0[4], s0, in0[4]),  (w1[0], select, in1[0]),
29           (w1[1], select, in1[1]), (w1[2], select, in1[2]),
30           (w1[3], select, in1[3]), (w1[4], select, in1[4]);
31         or #1 (out[0], w0[0], w1[0]), (out[1], w0[1], w1[1]),
32           (out[2], w0[2], w1[2]), (out[3], w0[3], w1[3]),
33           (out[4], w0[4], w1[4]);
34 endmodule // mux2
35
36 module mux2x32 (in0, in1, select, out);
37     input [31:0] in0,in1;
38     input select;
39     output [31:0] out;
40
41     wire          s0;
42     wire [31:0] w0,w1;
43
44         not #1 (s0, select);
45         and #1
46           (w0[0], s0, in0[0]), (w0[1], s0, in0[1]), (w0[2], s0, in0[2]),
47           (w0[3], s0, in0[3]), (w0[4], s0, in0[4]), (w0[5], s0, in0[5]),
48           (w0[6], s0, in0[6]), (w0[7], s0, in0[7]), (w0[8], s0, in0[8]),
49           (w0[9], s0, in0[9]), (w0[10], s0, in0[10]), (w0[11], s0, in0[11]),
50           (w0[12], s0, in0[12]), (w0[13], s0, in0[13]), (w0[14], s0, in0[14]),
51           (w0[15], s0, in0[15]), (w0[16], s0, in0[16]), (w0[17], s0, in0[17]),
52           (w0[18], s0, in0[18]), (w0[19], s0, in0[19]), (w0[20], s0, in0[20]),
53           (w0[21], s0, in0[21]), (w0[22], s0, in0[22]), (w0[23], s0, in0[23]),
54           (w0[24], s0, in0[24]), (w0[25], s0, in0[25]), (w0[26], s0, in0[26]),
55           (w0[27], s0, in0[27]), (w0[28], s0, in0[28]), (w0[29], s0, in0[29]),
56           (w0[30], s0, in0[30]), (w0[31], s0, in0[31]),
57
58           (w1[0], select, in1[0]), (w1[1], select, in1[1]),
59           (w1[2], select, in1[2]), (w1[3], select, in1[3]),
60           (w1[4], select, in1[4]), (w1[5], select, in1[5]),
61           (w1[6], select, in1[6]), (w1[7], select, in1[7]),
62           (w1[8], select, in1[8]), (w1[9], select, in1[9]),
63           (w1[10], select, in1[10]), (w1[11], select, in1[11]),
64           (w1[12], select, in1[12]), (w1[13], select, in1[13]),
65           (w1[14], select, in1[14]), (w1[15], select, in1[15]),
66           (w1[16], select, in1[16]), (w1[17], select, in1[17]),
67           (w1[18], select, in1[18]), (w1[19], select, in1[19]),
68           (w1[20], select, in1[20]), (w1[21], select, in1[21]),
69           (w1[22], select, in1[22]), (w1[23], select, in1[23]),
70           (w1[24], select, in1[24]), (w1[25], select, in1[25]),
71           (w1[26], select, in1[26]), (w1[27], select, in1[27]),
```

```
72          (w1[28], select, in1[28]), (w1[29], select, in1[29]),
73          (w1[30], select, in1[30]), (w1[31], select, in1[31]);
74        or #1
75          (out[0],w1[0], w0[0]), (out[1],w1[1], w0[1]), (out[2],w1[2], w0[2]),
76          (out[3],w1[3], w0[3]), (out[4],w1[4], w0[4]), (out[5],w1[5], w0[5]),
77          (out[6],w1[6], w0[6]), (out[7],w1[7], w0[7]), (out[8],w1[8], w0[8]),
78          (out[9],w1[9], w0[9]), (out[10],w1[10], w0[10]),
79          (out[11],w1[11], w0[11]), (out[12],w1[12], w0[12]),
80          (out[13],w1[13], w0[13]), (out[14],w1[14], w0[14]),
81          (out[15],w1[15], w0[15]), (out[16],w1[16], w0[16]),
82          (out[17],w1[17], w0[17]), (out[18],w1[18], w0[18]),
83          (out[19],w1[19], w0[19]), (out[20],w1[20], w0[20]),
84          (out[21],w1[21], w0[21]), (out[22],w1[22], w0[22]),
85          (out[23],w1[23], w0[23]), (out[24],w1[24], w0[24]),
86          (out[25],w1[25], w0[25]), (out[26],w1[26], w0[26]),
87          (out[27],w1[27], w0[27]), (out[28],w1[28], w0[28]),
88          (out[29],w1[29], w0[29]), (out[30],w1[30], w0[30]),
89          (out[31],w1[31], w0[31]);
90  endmodule // mux2x32
91
92
93  module mux4x32(in00, in01, in10, in11, select, out);
94     input [31:0] in00, in01, in10, in11;
95     input [1:0]         select;
96     output [31:0] out;
97
98     wire [31:0]          w0, w1;
99
100    mux2x32 m1(in00, in01, select[0], w0);
101    mux2x32 m2(in10, in11, select[0], w1);
102    mux2x32 m3(w0, w1, select[1], out);
103 endmodule // mux4x32
104
105 module mux8x32(in000, in001, in010, in011, in100, in101, in110, in111,
106                select, out);
107    input [31:0] in000, in001, in010, in011, in100, in101, in110, in111;
108    input [2:0]         select;
109    output [31:0] out;
110
111    wire [31:0]          w0, w1;
112
113    mux4x32 m1(in000, in001, in010, in011, select[1:0], w0);
114    mux4x32 m2(in100, in101, in110, in111, select[1:0], w1);
115    mux2x32 m3(w0, w1, select[2], out);
116 endmodule // 8x32
117
118 module mux16x32(in0000, in0001, in0010, in0011, in0100, in0101, in0110, in0111,
119                 in1000, in1001, in1010, in1011, in1100, in1101, in1110, in1111,
120                 select, out);
121    input[31:0] in0000, in0001, in0010, in0011, in0100, in0101, in0110, in0111,
122                in1000, in1001, in1010, in1011, in1100, in1101, in1110, in1111;
123    input [3:0] select;
```

```
124     output [31:0] out;
125
126     wire [31:0]          w0, w1;
127
128     mux8x32 m1(in0000, in0001, in0010, in0011, in0100, in0101, in0110, in0111,
129                select[2:0], w0);
130     mux8x32 m2(in1000, in1001, in1010, in1011, in1100, in1101, in1110, in1111,
131                select[2:0], w1);
132     mux2x32 m3(w0, w1, select[3], out);
133  endmodule // mux16x32
134
135  module mux32x32(regs00000, regs00001, regs00010, regs00011,
136                  regs00100, regs00101, regs00110, regs00111,
137                  regs01000, regs01001, regs01010, regs01011,
138                  regs01100, regs01101, regs01110, regs01111,
139                  regs10000, regs10001, regs10010, regs10011,
140                  regs10100, regs10101, regs10110, regs10111,
141                  regs11000, regs11001, regs11010, regs11011,
142                  regs11100, regs11101, regs11110, regs11111,
143                  select, out);
144
145     input [31:0] regs00000, regs00001, regs00010, regs00011,
146                  regs00100, regs00101, regs00110, regs00111,
147                  regs01000, regs01001, regs01010, regs01011,
148                  regs01100, regs01101, regs01110, regs01111,
149                  regs10000, regs10001, regs10010, regs10011,
150                  regs10100, regs10101, regs10110, regs10111,
151                  regs11000, regs11001, regs11010, regs11011,
152                  regs11100, regs11101, regs11110, regs11111;
153     input [4:0]         select;
154     output [31:0] out;
155
156     wire [31:0]          w0, w1;
157
158     mux16x32 m1(regs00000, regs00001, regs00010, regs00011,
159                 regs00100, regs00101, regs00110, regs00111,
160                 regs01000, regs01001, regs01010, regs01011,
161                 regs01100, regs01101, regs01110, regs01111,
162                 select[3:0], w0);
163     mux16x32 m2(regs10000, regs10001, regs10010, regs10011,
164                 regs10100, regs10101, regs10110, regs10111,
165                 regs11000, regs11001, regs11010, regs11011,
166                 regs11100, regs11101, regs11110, regs11111,
167                 select[3:0], w1);
168     mux2x32 m3(w0, w1, select[4], out);
169  endmodule // mux32x32
170
171
172
173  module mux4(in0, in1, in2, in3, select, out);
174     input in0, in1, in2, in3;
175     input [1:0] select;
```

```
176     output out;
177     wire   w01, w23;
178
179     mux2 m01(in0, in1, select[0], w01);
180     mux2 m23(in2, in3, select[0], w23);
181
182     mux2 mOUT(w01, w23, select[1], out);
183  endmodule // mux4
```

```
1   // Register Modules
2   // Synchronous Version
3   // Robert Webb
4
5   module Dlatch(Data, En, Q);
6      input Data, En;
7      output Q;
8
9      reg           D;
10
11     always #1
12       if (Data)
13         D = Data;
14       else
15         D = 0;
16
17     wire   enD ,enNotD;
18     wire   notnotQ, notD;
19
20     not  (notD, D),
21       (notEnNotD, enNotD);
22     and #2 (enD, D, En),
23       (enNotD, notD, En);
24     or #1 (notnotQ, enD, Q);
25     and #2 (Q, notnotQ, notEnNotD);
26  endmodule // Dlatch
27
28  module reg32(D, En, clk, Q);
29     input [31:0] D;
30     input         En, clk;
31     output [31:0] Q;
32
33     wire          Enr, trig;
34
35     PosTrigger p(clk, trig);
36
37     and #1 (Enr, En, trig);
38
39     Dlatch r0(D[0], Enr, Q[0]); Dlatch r1(D[1], Enr, Q[1]);
40     Dlatch r2(D[2], Enr, Q[2]); Dlatch r3(D[3], Enr, Q[3]);
41     Dlatch r4(D[4], Enr, Q[4]); Dlatch r5(D[5], Enr, Q[5]);
42     Dlatch r6(D[6], Enr, Q[6]); Dlatch r7(D[7], Enr, Q[7]);
43     Dlatch r8(D[8], Enr, Q[8]); Dlatch r9(D[9], Enr, Q[9]);
```

```
44     Dlatch r10(D[10], Enr, Q[10]); Dlatch r11(D[11], Enr, Q[11]);
45     Dlatch r12(D[12], Enr, Q[12]); Dlatch r13(D[13], Enr, Q[13]);
46     Dlatch r14(D[14], Enr, Q[14]); Dlatch r15(D[15], Enr, Q[15]);
47     Dlatch r16(D[16], Enr, Q[16]); Dlatch r17(D[17], Enr, Q[17]);
48     Dlatch r18(D[18], Enr, Q[18]); Dlatch r19(D[19], Enr, Q[19]);
49     Dlatch r20(D[20], Enr, Q[20]); Dlatch r21(D[21], Enr, Q[21]);
50     Dlatch r22(D[22], Enr, Q[22]); Dlatch r23(D[23], Enr, Q[23]);
51     Dlatch r24(D[24], Enr, Q[24]); Dlatch r25(D[25], Enr, Q[25]);
52     Dlatch r26(D[26], Enr, Q[26]); Dlatch r27(D[27], Enr, Q[27]);
53     Dlatch r28(D[28], Enr, Q[28]); Dlatch r29(D[29], Enr, Q[29]);
54     Dlatch r30(D[30], Enr, Q[30]); Dlatch r31(D[31], Enr, Q[31]);
55  endmodule // reg32
56
57  module PosTrigger(clk, trig);
58     input clk;
59     output trig;
60
61     wire   notclk;
62
63     not #3  (notclk, clk);
64     and  (trig, clk, notclk);
65  endmodule // PosTrigger
66
67  module NegTrigger(clk, trig);
68     input clk;
69     output trig;
70
71     wire   notclk;
72     wire   notnotclk;
73
74     not #1 (notclk, clk),
75       (notnotclk, notclk);
76     and (trig, notclk, notnotclk);
77  endmodule // NegTrigger
78
79  module registerfile(outreg1, outreg2, inreg, inv, out1, out2, clk, rst);
80     input [4:0] outreg1, outreg2, inreg;
81     input [31:0] inv;
82     input         clk, rst;
83     output [31:0] out1, out2;
84
85     wire [31:0]          e;
86     wire [31:0] r0, r1, r2, r3, r4, r5, r6, r7,
87                 r8, r9, r10, r11, r12, r13, r14, r15,
88                 r16, r17, r18, r19, r20, r21, r22, r23,
89                 r24, r25, r26, r27, r28, r29, r30, r31;
90
91     always @ (posedge clk)
92       begin
93          if (rst)
94               begin
95                     $strobe("Wrote %d to register $%d", inv, inreg);
```

```
96              end
97       end
98
99     enable en(inreg, rst, e);
100
101    reg32 rg0(32'b0, 1, clk, r0); reg32 rg1(inv, e[1], clk, r1);
102    reg32 rg2(inv, e[2], clk, r2); reg32 rg3(inv, e[3], clk, r3);
103    reg32 rg4(inv, e[4], clk, r4); reg32 rg5(inv, e[5], clk, r5);
104    reg32 rg6(inv, e[6], clk, r6); reg32 rg7(inv, e[7], clk, r7);
105    reg32 rg8(inv, e[8], clk, r8); reg32 rg9(inv, e[9], clk, r9);
106    reg32 rg10(inv, e[10], clk, r10); reg32 rg11(inv, e[11], clk, r11);
107    reg32 rg12(inv, e[12], clk, r12); reg32 rg13(inv, e[13], clk, r13);
108    reg32 rg14(inv, e[14], clk, r14); reg32 rg15(inv, e[15], clk, r15);
109    reg32 rg16(inv, e[16], clk, r16); reg32 rg17(inv, e[17], clk, r17);
110    reg32 rg18(inv, e[18], clk, r18); reg32 rg19(inv, e[19], clk, r19);
111    reg32 rg20(inv, e[20], clk, r20); reg32 rg21(inv, e[21], clk, r21);
112    reg32 rg22(inv, e[22], clk, r22); reg32 rg23(inv, e[23], clk, r23);
113    reg32 rg24(inv, e[24], clk, r24); reg32 rg25(inv, e[25], clk, r25);
114    reg32 rg26(inv, e[26], clk, r26); reg32 rg27(inv, e[27], clk, r27);
115    reg32 rg28(inv, e[28], clk, r28); reg32 rg29(inv, e[29], clk, r29);
116    reg32 rg30(inv, e[30], clk, r30); reg32 rg31(inv, e[31], clk, r31);
117
118    mux32x32 m1(r0, r1, r2, r3, r4, r5, r6, r7,
119                r8, r9, r10, r11, r12, r13, r14, r15,
120                r16, r17, r18, r19, r20, r21, r22, r23,
121                r24, r25, r26, r27, r28, r29, r30, r31,
122                outreg1, out1);
123
124    mux32x32 m2(r0, r1, r2, r3, r4, r5, r6, r7,
125                r8, r9, r10, r11, r12, r13, r14, r15,
126                r16, r17, r18, r19, r20, r21, r22, r23,
127                r24, r25, r26, r27, r28, r29, r30, r31,
128                outreg2, out2);
129
130 endmodule // registerfile
```

```
1  // Instruction Fetch Module
2  // Synchronous Version
3  // Robert Webb
4
5  module instructionfetch(pc, clk, instruction);
6     input [31:0] pc;
7     input         clk;
8     output [31:0] instruction;
9
10    reg [31:0]          instructions[0:56];
11    reg [31:0]          instruction;
12
13    always @ (posedge clk) #10
14      begin
15         if (pc > 60) $stop;
16         instruction = instructions[pc];
```

```
17        end
18
19      initial
20        begin
21        instructions[0] = 0;
22        instruction = instructions[0];
23        instructions[4] = 32'b00100000000001010000000000000001; // addi $5, $0, 1
24        instructions[8] = 32'b00100000000001000000000000010100; // addi $4,$0,20
25        instructions[12] = 32'b10101100000000000000000000000000; // sw $0, $0(0)
26        instructions[16] = 32'b00100000000000010000000000000001; //addi $1, $0, 1
27        instructions[20] = 32'b10101100000000010000000000000100; //sw $1, $0(4)
28        //Loop
29        instructions[24] = 32'b10001100000001100000000000000000; //lw $6, $0(0)
30        instructions[28] = 32'b10001100000001110000000000000100; //lw $7, $0(4)
31        instructions[32] = 32'b00000000111001100100000000100000;//add $8, $7, $6
32        instructions[36] = 32'b10101100000001110000000000000000; //sw $7, $0(0)
33        instructions[40] = 32'b10101100000001000000000000000100; //sw $8, $0(4)
34        instructions[44] = 32'b00100000100001001111111111111111; //addi $4, $4, -1
35        instructions[48] = 32'b00010100101001001111111111111001;// bne $4,$5,-7
36        //EndLoop
37        instructions[52] = 32'b00000001000000000001000000100000;// add $2,$8,$0
38        instructions[56] = 32'b00001000000000000000000000001101; // j 52
39        instructions[60] = 32'b0;
40        end // initial begin
41    endmodule // instructionfetch
```

```
1   // Memory Module
2   // Synchronous Version
3   // Robert Webb
4
5   module memory(readdata, address, writedata, memwrite, memread, clk);
6      input [31:0] address, writedata;
7      input         memread, memwrite, clk;
8      output [31:0] readdata;
9
10     reg [31:0]          readdata;
11     reg [31:0]          sys[127:0];
12
13     always @ (negedge clk)
14       begin #5
15          if (memread)
16            begin
17               readdata = sys[address];
18               $strobe("Read %d from address %d \n", readdata, address);
19            end
20       end
21
22     always @ (negedge clk)
23       begin #5
24          if (memwrite)
25            begin
26               sys[address] = writedata;
```

```
27              $strobe("Wrote %d to address $%d \n", writedata, address);
28            end
29       end
30
31     initial
32       begin
33          sys[0] = 32'b0;
34          readdata = 0;
35       end
36  endmodule // memory
```

```
1   // Control Modules
2   // Synchronous Version
3   // Robert Webb
4
5   module control(inst, CLK, regDist, Branch, Jump, ALUop, ALUSrc, regWrite,
6                  MemWrite, MemRead, MemtoReg);
7      input[5:0] inst;
8      input      CLK;
9
10     output     regDist, Branch, ALUSrc, regWrite,
11                MemRead, MemWrite, MemtoReg, Jump;
12     output [1:0] ALUop;
13
14     reg     regDist, Branch, ALUSrc,regWrite,
15                MemRead, MemWrite, MemtoReg, Jump;
16     reg [1:0] ALUop;
17
18     initial
19       begin
20          regWrite = 0; regDist = 0; Branch = 0; Jump = 0;
21          ALUop = 2'b00; ALUSrc = 0;
22          MemRead = 0; MemWrite = 0; MemtoReg = 0;
23       end
24
25     always
26     begin #1
27     if(inst == 6'b0) //addi
28       begin
29          regWrite = 1; regDist = 1; Branch = 0; Jump = 0;
30          ALUop = 2'b10; ALUSrc = 0;
31          MemRead = 0; MemWrite = 0; MemtoReg = 0;
32          //$strobe("R-Type Instruction");
33       end // if (inst == 6'b0)
34
35     if(inst == 6'b001000) //addi
36       begin
37          regWrite = 1; regDist = 0; Branch = 0; Jump = 0;
38          ALUop = 2'b00; ALUSrc = 1;
39          MemWrite = 0; MemRead = 0; MemtoReg = 0;
40          //$strobe("addi");
41       end // if (inst == 6'b001000)
```

```
42
43      if(inst == 6'b000101) //bne
44       begin
45          regWrite = 0; regDist = 0; Branch = 1; Jump = 0;
46          ALUop = 2'b01; ALUSrc = 0;
47          MemWrite = 0; MemRead = 0; MemtoReg = 0;
48          //$strobe("bne");
49       end
50
51     if(inst == 6'b000010) //j
52       begin
53          regWrite = 0; regDist = 0; Branch = 0; Jump = 1;
54          ALUop = 2'b01; ALUSrc = 0;
55          MemWrite = 0; MemRead = 0; MemtoReg = 0;
56          //$strobe("jump");
57       end
58
59     if(inst == 6'b100011) //lw
60       begin
61          regWrite = 1; regDist = 0; Branch = 0; Jump = 0;
62          ALUop = 2'b00; ALUSrc = 1;
63          MemWrite = 0; MemRead = 1; MemtoReg = 1;
64          //$strobe("load word");
65       end
66
67     if(inst == 6'b101011) //sw
68       begin
69          regWrite = 0; regDist = 0; Branch = 0; Jump = 0;
70          ALUop = 2'b00; ALUSrc = 1;
71          MemWrite = 1; MemRead = 0; MemtoReg = 0;
72          //$strobe("store word");
73       end // if (inst == 6'b001000)
74      end // always begin
75  endmodule // control
76
77  module ALUcontrol(inst, ContOP, op);
78     input[5:0] inst;
79     input [1:0]     ContOP;
80     output [2:0] op;
81
82     reg [2:0]         op;
83
84     initial
85       op = 0;
86     always
87       begin #1
88         if(ContOP == 2'b00)
89           op = 3'b010;  // addi lw sw
90         if(ContOP == 2'b01)
91           op = 3'b110;  // bne
92         if(ContOP == 2'b10)  //R-type instructions
93           begin
```

```
94              if(inst == 6'b100000)
95                op = 3'b010;  //add
96              if(inst == 6'b100100)
97                op = 3'b000;  // and
98              if(inst == 6'b100010)
99                op = 3'b110;  // sub
100             if(inst == 6'b100101)
101               op = 3'b001;  // or
102             if(inst == 6'b101010)
103               op = 3'b111;  //slt
104         end // if (ContOP = 2'b10)
105     end // always begin
106 endmodule // ALUcontrol
```

```
1  // ALU Modules
2  // Synchronous Version
3  // Robert Webb
4
5  module adder1(a, b, cin, cout, result);
6     input a, b, cin;
7     output result, cout;
8     wire   w0,w1,w2, na, nb, nc, a1, a2, a3, a4, a5, a6, o7, o8;
9
10    and #1 (w0, a, cin),
11      (w1, b, cin),
12      (w2, a, b);
13    or #1 (cout, w0, w1, w2);
14    not #1 (na, a), (nb, b),
15      (nc, cin);
16    and #1 (a1, nb, nc),
17      (a2, b, cin);
18    or #1 (o7, a1, a2);
19    and #1 (a3, b, nc),
20      (a4, cin, nb);
21    or #1 (o8, a3, a4);
22    and #1 (a5, o7, a),
23      (a6, na, o8);
24    or #1 (result, a5, a6);
25 endmodule // adder1
26
27 module alu1(a,b,cin, less, binv, op, cout, result);
28    input a,b,cin,less,binv;
29    input [1:0]         op;
30    output      result, cout;
31
32    wire        nb, b1, w0, w1, set;
33
34    not #1 (nb, b);
35    mux2 bmux(b, nb, binv, b1);
36    and #1 (w0, a, b1);
37    or #1 (w1, a, b1);
38    adder1 add(a, b1, cin, cout, set);
```

```
39     mux4 mOUT(w0, w1, set, less, op, result);
40  endmodule // alu1
41
42  module alu1END(a,b,cin, less, binv, op, cout, result, set);
43     input a,b,cin,less,binv;
44     input [1:0] op;
45     output      result, cout, set;
46
47     wire        nb, b1, w0, w1;
48
49     not #1 (nb, b);
50     mux2 bmux(b, nb, binv, b1);
51     and #1 (w0, a, b1);
52     or #1 (w1, a, b1);
53     adder1 add(a, b1, cin, cout, set);
54     mux4 mOUT(w0, w1, set, less, op, result);
55  endmodule // alu1END
56
57
58  module alu32(a,b,cin, op, result, notzero);
59     input [31:0] a, b;
60     input [2:0]         op; // 000 = and, 001 = or, 010 = add, 110 = sub, 111 = slt
61     input         cin;
62
63     output         notzero;
64     output [31:0] result;
65
66     wire [30:0] w;
67     wire        less;
68     wire        overflow;
69
70     alu1 a0(a[0], b[0], op[2], less, op[2], op[1:0], w[0], result[0]);
71     alu1 a1(a[1], b[1], w[0], 1'b0, op[2], op[1:0], w[1], result[1]);
72     alu1 a2(a[2], b[2], w[1], 1'b0, op[2], op[1:0], w[2], result[2]);
73     alu1 a3(a[3], b[3], w[2], 1'b0, op[2], op[1:0], w[3], result[3]);
74     alu1 a4(a[4], b[4], w[3], 1'b0, op[2], op[1:0], w[4], result[4]);
75     alu1 a5(a[5], b[5], w[4], 1'b0, op[2], op[1:0], w[5], result[5]);
76     alu1 a6(a[6], b[6], w[5], 1'b0, op[2], op[1:0], w[6], result[6]);
77     alu1 a7(a[7], b[7], w[6], 1'b0, op[2], op[1:0], w[7], result[7]);
78     alu1 a8(a[8], b[8], w[7], 1'b0, op[2], op[1:0], w[8], result[8]);
79     alu1 a9(a[9], b[9], w[8], 1'b0, op[2], op[1:0], w[9], result[9]);
80     alu1 a10(a[10], b[10], w[9], 1'b0, op[2], op[1:0], w[10], result[10]);
81     alu1 a11(a[11], b[11], w[10], 1'b0, op[2], op[1:0], w[11], result[11]);
82     alu1 a12(a[12], b[12], w[11], 1'b0, op[2], op[1:0], w[12], result[12]);
83     alu1 a13(a[13], b[13], w[12], 1'b0, op[2], op[1:0], w[13], result[13]);
84     alu1 a14(a[14], b[14], w[13], 1'b0, op[2], op[1:0], w[14], result[14]);
85     alu1 a15(a[15], b[15], w[14], 1'b0, op[2], op[1:0], w[15], result[15]);
86     alu1 a16(a[16], b[16], w[15], 1'b0, op[2], op[1:0], w[16], result[16]);
87     alu1 a17(a[17], b[17], w[16], 1'b0, op[2], op[1:0], w[17], result[17]);
88     alu1 a18(a[18], b[18], w[17], 1'b0, op[2], op[1:0], w[18], result[18]);
89     alu1 a19(a[19], b[19], w[18], 1'b0, op[2], op[1:0], w[19], result[19]);
90     alu1 a20(a[20], b[20], w[19], 1'b0, op[2], op[1:0], w[20], result[20]);
```

```
    alu1 a21(a[21], b[21], w[20], 1'b0, op[2], op[1:0], w[21], result[21]);
    alu1 a22(a[22], b[22], w[21], 1'b0, op[2], op[1:0], w[22], result[22]);
    alu1 a23(a[23], b[23], w[22], 1'b0, op[2], op[1:0], w[23], result[23]);
    alu1 a24(a[24], b[24], w[23], 1'b0, op[2], op[1:0], w[24], result[24]);
    alu1 a25(a[25], b[25], w[24], 1'b0, op[2], op[1:0], w[25], result[25]);
    alu1 a26(a[26], b[26], w[25], 1'b0, op[2], op[1:0], w[26], result[26]);
    alu1 a27(a[27], b[27], w[26], 1'b0, op[2], op[1:0], w[27], result[27]);
    alu1 a28(a[28], b[28], w[27], 1'b0, op[2], op[1:0], w[28], result[28]);
    alu1 a29(a[29], b[29], w[28], 1'b0, op[2], op[1:0], w[29], result[29]);
    alu1 a30(a[30], b[30], w[29], 1'b0, op[2], op[1:0], w[30], result[30]);
    alu1END a31(a[31], b[30], w[30], 1'b0, op[2], op[1:0], overflow,
                result[31], less);
    or #1 (notzero, result[31], result[30], result[29], result[28], result[27],
                  result[26], result[25], result[24], result[23], result[22],
                  result[21], result[20], result[19], result[18], result[17],
                  result[16], result[15], result[14], result[13], result[12],
                  result[11], result[10], result[9], result[8], result[7],
                  result[6], result[5], result[4], result[3], result[2],
                  result[1], result[0]);
endmodule // alu32


// Synchronous Version
// Robert Webb

module CPU(clk, result, pc);
   input clk;
   output [31:0] result;
   output [31:0] pc;

   wire [31:0] PC, PCp4, sgnExtIM, sgnExtIMx4, branchPC, nextPC, data1, data2,
               immed, aluResult, memResult, finalout, inst, JumpPC, finalNextPC;
   wire        RegWrite, isBranch, takeJump, notzero, takeBranch,
               ALUsrc, RegDist, memwrite, memread, memtoreg,
               nowhere1, nowhere2;
   wire [1:0]  ALUop;
   wire [2:0]  RealALUop;
   wire [4:0]  WriteReg;

   reg32 programc(finalNextPC, 1'b1, clk, PC);
   alu32 addpc(PC, 32'd4, 1'b0, 3'b010, PCp4, nowhere1);
   signextend16to32 se(inst[15:0], sgnExtIM);
   shiftleft2 sl(sgnExtIM, sgnExtIMx4);
   alu32 bPC(PCp4, sgnExtIMx4, 1'b0, 3'b010, branchPC, nowhere2);
   and #1 (takeBranch, isBranch, notzero);
   mux2x32 PCchooser(PCp4, branchPC, takeBranch, nextPC);
   jumpPC jumpPCcalc(PC[31:28], inst[25:0], JumpPC);
   mux2x32 JumpChooser(nextPC, JumpPC, takeJump, finalNextPC);
   instructionfetch InsF(PC, clk, inst);
   control controlU(inst[31:26], clk, RegDist, isBranch, takeJump, ALUop, ALUsrc,
                    RegWrite, memwrite, memread, memtoreg);
   ALUcontrol ALUcontrolI(inst[5:0], ALUop, RealALUop);
   mux2x5 WriteChooser(inst[20:16], inst[15:11], RegDist, WriteReg);
```

```
32    registerfile regfile(inst[25:21], inst[20:16], WriteReg,
33                         finalout, data1, data2, clk, RegWrite);
34    mux2x32 dataChooser(data2, sgnExtIM, ALUsrc, immed);
35    alu32 RealALU(data1, immed, 1'b0, RealALUop, aluResult, notzero);
36    memory memunit(memResult, aluResult, data2, memwrite, memread, clk);
37    mux2x32 memORalu(aluResult, memResult, memtoreg, finalout);
38    assign result = finalNextPC;
39    assign pc = PC;
40 endmodule // CPU
41
42 module main;
43    reg                  CLK;
44    wire [31:0] pc, result;
45
46    CPU SynchronousSingleCycle(CLK, result, pc);
47    initial
48      begin
49         CLK = 1'b0;
50         forever
51           #48 CLK = ~CLK;
52      end
53    initial
54      begin
55         #14000 $stop;
56      end
57    initial
58      $monitor( "pc=%d result=%d time=%d CLK=%b",
59                pc, result, $time, CLK);
60 endmodule // main
```

Appendix B

The Complete Single-Cycle Asynchronous Source Code

```
1  // Simple Dual-Rail Gates
2  // Asynchronous Processor
3  // Robert Webb
4
5  module not2(outH, outL, inH, inL);
6     input inH,inL;
7     output outH, outL;
8
9     buf (outH, inL),
10      (outL, inH);
11 endmodule // not2
12
13 module and2(outH, outL, AH, AL, BH, BL);
14    input AH,AL,BH,BL;
15    output outH, outL;
16
17    and #(1,2) (outH, AH, BH);
18    or #(0,1) (outL, AL, BL);
19 endmodule // and2
20
21 module or2(outH, outL, AH, AL, BH, BL);
22    input AH,AL,BH,BL;
23    output outH, outL;
24
25    or #(0,1) (outH, AH, BH);
26    and #(1,2) (outL, AL, BL);
27 endmodule // or2
```

```
1  // Buffer Modules
2  // Asynchronous Processor
3  // Robert Webb
4
5  module enable(i, rst, enbit);
6     input [4:0] i;
7     input       rst;
8     output [31:0] enbit;
9
10    wire [4:0] ni;
11
12    not (ni[0], i[0]), (ni[1], i[1]), (ni[2], i[2]),
13      (ni[3], i[3]), (ni[4], i[4]);
14
15    and #1 (enbit[0], ni[4], ni[3], ni[2], ni[1], ni[0], rst),
16      (enbit[1], ni[4], ni[3], ni[2], ni[1], i[0], rst),
17      (enbit[2], ni[4], ni[3], ni[2], i[1], ni[0], rst),
18      (enbit[3], ni[4], ni[3], ni[2], i[1], i[0], rst),
19      (enbit[4], ni[4], ni[3], i[2], ni[1], ni[0], rst),
20      (enbit[5], ni[4], ni[3], i[2], ni[1], i[0], rst),
21      (enbit[6], ni[4], ni[3], i[2], i[1], ni[0], rst),
22      (enbit[7], ni[4], ni[3], i[2], i[1], i[0], rst),
23      (enbit[8], ni[4], i[3], ni[2], ni[1], ni[0], rst),
24      (enbit[9], ni[4], i[3], ni[2], ni[1], i[0], rst),
25      (enbit[10], ni[4], i[3], ni[2], i[1], ni[0], rst),
26      (enbit[11], ni[4], i[3], ni[2], i[1], i[0], rst),
27      (enbit[12], ni[4], i[3], i[2], ni[1], ni[0], rst),
28      (enbit[13], ni[4], i[3], i[2], ni[1], i[0], rst),
29      (enbit[14], ni[4], i[3], i[2], i[1], ni[0], rst),
30      (enbit[15], ni[4], i[3], i[2], i[1], i[0], rst),
31      (enbit[16], i[4], ni[3], ni[2], ni[1], ni[0], rst),
32      (enbit[17], i[4], ni[3], ni[2], ni[1], i[0], rst),
33      (enbit[18], i[4], ni[3], ni[2], i[1], ni[0], rst),
34      (enbit[19], i[4], ni[3], ni[2], i[1], i[0], rst),
35      (enbit[20], i[4], ni[3], i[2], ni[1], ni[0], rst),
36      (enbit[21], i[4], ni[3], i[2], ni[1], i[0], rst),
37      (enbit[22], i[4], ni[3], i[2], i[1], ni[0], rst),
38      (enbit[23], i[4], ni[3], i[2], i[1], i[0], rst),
39      (enbit[24], i[4], i[3], ni[2], ni[1], ni[0], rst),
40      (enbit[25], i[4], i[3], ni[2], ni[1], i[0], rst),
41      (enbit[26], i[4], i[3], ni[2], i[1], ni[0], rst),
42      (enbit[27], i[4], i[3], ni[2], i[1], i[0], rst),
43      (enbit[28], i[4], i[3], i[2], ni[1], ni[0], rst),
44      (enbit[29], i[4], i[3], i[2], ni[1], i[0], rst),
45      (enbit[30], i[4], i[3], i[2], i[1], ni[0], rst),
46      (enbit[31], i[4], i[3], i[2], i[1], i[0], rst);
47 endmodule // enable
48
49 module signextend16to32(immH, immL, outH, outL);
50    input [15:0] immH, immL;
51    output [31:0] outH, outL;
52
```

```
53      buf
54        (outL[0], immL[0]), (outL[1], immL[1]), (outL[2], immL[2]),
55        (outL[3], immL[3]), (outL[4], immL[4]), (outL[5], immL[5]),
56        (outL[6], immL[6]), (outL[7], immL[7]), (outL[8], immL[8]),
57        (outL[9], immL[9]), (outL[10], immL[10]), (outL[11], immL[11]),
58        (outL[12], immL[12]), (outL[13], immL[13]), (outL[14], immL[14]),
59        (outL[15], immL[15]), (outL[16], immL[15]), (outL[17], immL[15]),
60        (outL[18], immL[15]), (outL[19], immL[15]), (outL[20], immL[15]),
61        (outL[21], immL[15]), (outL[22], immL[15]), (outL[23], immL[15]),
62        (outL[24], immL[15]), (outL[25], immL[15]), (outL[26], immL[15]),
63        (outL[27], immL[15]), (outL[28], immL[15]), (outL[29], immL[15]),
64        (outL[30], immL[15]), (outL[31], immL[15]),
65
66        (outH[0], immH[0]), (outH[1], immH[1]), (outH[2], immH[2]),
67        (outH[3], immH[3]), (outH[4], immH[4]), (outH[5], immH[5]),
68        (outH[6], immH[6]), (outH[7], immH[7]), (outH[8], immH[8]),
69        (outH[9], immH[9]), (outH[10], immH[10]), (outH[11], immH[11]),
70        (outH[12], immH[12]), (outH[13], immH[13]), (outH[14], immH[14]),
71        (outH[15], immH[15]), (outH[16], immH[15]), (outH[17], immH[15]),
72        (outH[18], immH[15]), (outH[19], immH[15]), (outH[20], immH[15]),
73        (outH[21], immH[15]), (outH[22], immH[15]), (outH[23], immH[15]),
74        (outH[24], immH[15]), (outH[25], immH[15]), (outH[26], immH[15]),
75        (outH[27], immH[15]), (outH[28], immH[15]), (outH[29], immH[15]),
76        (outH[30], immH[15]), (outH[31], immH[15]);
77  endmodule // signextend16to32
78
79  module shiftleft2(inpH, inpL, outH, outL);
80     input [31:0] inpH, inpL;
81     output [31:0] outH, outL;
82
83     assign outH[0] = 1'b0, outH[1] = 1'b0;
84     assign outL[0] = 1'b1, outL[1] = 1'b1;
85     buf
86       (outL[2], inpL[0]), (outL[3], inpL[1]), (outL[4], inpL[2]),
87       (outL[5], inpL[3]), (outL[6], inpL[4]), (outL[7], inpL[5]),
88       (outL[8], inpL[6]), (outL[9], inpL[7]), (outL[10], inpL[8]),
89       (outL[11], inpL[9]), (outL[12], inpL[10]), (outL[13], inpL[11]),
90       (outL[14], inpL[12]), (outL[15], inpL[13]), (outL[16], inpL[14]),
91       (outL[17], inpL[15]), (outL[18], inpL[16]), (outL[19], inpL[17]),
92       (outL[20], inpL[18]), (outL[21], inpL[19]), (outL[22], inpL[20]),
93       (outL[23], inpL[21]), (outL[24], inpL[22]), (outL[25], inpL[23]),
94       (outL[26], inpL[24]), (outL[27], inpL[25]), (outL[28], inpL[26]),
95       (outL[29], inpL[27]), (outL[30], inpL[28]), (outL[31], inpL[29]),
96
97       (outH[2], inpH[0]), (outH[3], inpH[1]), (outH[4], inpH[2]),
98       (outH[5], inpH[3]), (outH[6], inpH[4]), (outH[7], inpH[5]),
99       (outH[8], inpH[6]), (outH[9], inpH[7]), (outH[10], inpH[8]),
100      (outH[11], inpH[9]), (outH[12], inpH[10]), (outH[13], inpH[11]),
101      (outH[14], inpH[12]), (outH[15], inpH[13]), (outH[16], inpH[14]),
102      (outH[17], inpH[15]), (outH[18], inpH[16]), (outH[19], inpH[17]),
103      (outH[20], inpH[18]), (outH[21], inpH[19]), (outH[22], inpH[20]),
104      (outH[23], inpH[21]), (outH[24], inpH[22]), (outH[25], inpH[23]),
```

```
105       (outH[26], inpH[24]), (outH[27], inpH[25]), (outH[28], inpH[26]),
106       (outH[29], inpH[27]), (outH[30], inpH[28]), (outH[31], inpH[29]);
107   endmodule // shiftleft2
108
109   module jumpPC(topH, topL, inpH, inpL, outH, outL);
110      input [3:0] topH, topL;
111      input [25:0] inpH, inpL;
112
113      output [31:0] outH, outL;
114
115      assign outH[0] = 1'b0, outH[1] = 1'b0;
116      assign outL[0] = 1'b1, outL[1] = 1'b1;
117      buf
118        (outL[2], inpL[0]), (outL[3], inpL[1]), (outL[4], inpL[2]),
119        (outL[5], inpL[3]), (outL[6], inpL[4]), (outL[7], inpL[5]),
120        (outL[8], inpL[6]), (outL[9], inpL[7]), (outL[10], inpL[8]),
121        (outL[11], inpL[9]), (outL[12], inpL[10]), (outL[13], inpL[11]),
122        (outL[14], inpL[12]), (outL[15], inpL[13]), (outL[16], inpL[14]),
123        (outL[17], inpL[15]), (outL[18], inpL[16]), (outL[19], inpL[17]),
124        (outL[20], inpL[18]), (outL[21], inpL[19]), (outL[22], inpL[20]),
125        (outL[23], inpL[21]), (outL[24], inpL[22]), (outL[25], inpL[23]),
126        (outL[26], inpL[24]), (outL[27], inpL[25]), (outL[28], topL[0]),
127        (outL[29], topL[1]), (outL[30], topL[2]), (outL[31], topL[3]),
128
129        (outH[2], inpH[0]), (outH[3], inpH[1]), (outH[4], inpH[2]),
130        (outH[5], inpH[3]), (outH[6], inpH[4]), (outH[7], inpH[5]),
131        (outH[8], inpH[6]), (outH[9], inpH[7]), (outH[10], inpH[8]),
132        (outH[11], inpH[9]), (outH[12], inpH[10]), (outH[13], inpH[11]),
133        (outH[14], inpH[12]), (outH[15], inpH[13]), (outH[16], inpH[14]),
134        (outH[17], inpH[15]), (outH[18], inpH[16]), (outH[19], inpH[17]),
135        (outH[20], inpH[18]), (outH[21], inpH[19]), (outH[22], inpH[20]),
136        (outH[23], inpH[21]), (outH[24], inpH[22]), (outH[25], inpH[23]),
137        (outH[26], inpH[24]), (outH[27], inpH[25]), (outH[28], topH[0]),
138        (outH[29], topH[1]), (outH[30], topH[2]), (outH[31], topH[3]);
139   endmodule // jumpPC
```

```
1   // Multiplexor Modules
2   // Asynchronous Processor
3   // Robert Webb
4
5   module mux2 (in0H, in0L, in1H, in1L, select, outH, outL);
6          input in0H, in0L, in1H, in1L, select;
7          output outH, outL;
8          wire s0,w0H, w0L, w1H, w1L;
9
10         not (s0, select);
11
12         and2 h(w0H, w0L, s0, select, in0H, in0L),
13           l(w1H, w1L, select, s0, in1H, in1L);
14
15         or2 f(outH, outL, w0H, w0L, w1H, w1L);
16  endmodule // mux2
```

```
17
18 module mux2x5 (in0H, in0L, in1H, in1L, select, outH, outL);
19    input[4:0] in0H, in0L,in1H, in1L;
20    input select;
21    output[4:0] outH, outL;
22
23    mux2 m0(in0H[0], in0L[0], in1H[0], in1L[0], select, outH[0], outL[0]);
24    mux2 m1(in0H[1], in0L[1], in1H[1], in1L[1], select, outH[1], outL[1]);
25    mux2 m2(in0H[2], in0L[2], in1H[2], in1L[2], select, outH[2], outL[2]);
26    mux2 m3(in0H[3], in0L[3], in1H[3], in1L[3], select, outH[3], outL[3]);
27    mux2 m4(in0H[4], in0L[4], in1H[4], in1L[4], select, outH[4], outL[4]);
28 endmodule // mux2x5
29
30
31 module mux2x32 (in0H, in0L, in1H, in1L, select, outH, outL);
32    input[31:0] in0H, in0L,in1H, in1L;
33    input select;
34    output[31:0] outH, outL;
35
36    mux2 m0(in0H[0], in0L[0], in1H[0], in1L[0], select, outH[0], outL[0]);
37    mux2 m1(in0H[1], in0L[1], in1H[1], in1L[1], select, outH[1], outL[1]);
38    mux2 m2(in0H[2], in0L[2], in1H[2], in1L[2], select, outH[2], outL[2]);
39    mux2 m3(in0H[3], in0L[3], in1H[3], in1L[3], select, outH[3], outL[3]);
40    mux2 m4(in0H[4], in0L[4], in1H[4], in1L[4], select, outH[4], outL[4]);
41    mux2 m5(in0H[5], in0L[5], in1H[5], in1L[5], select, outH[5], outL[5]);
42    mux2 m6(in0H[6], in0L[6], in1H[6], in1L[6], select, outH[6], outL[6]);
43    mux2 m7(in0H[7], in0L[7], in1H[7], in1L[7], select, outH[7], outL[7]);
44    mux2 m8(in0H[8], in0L[8], in1H[8], in1L[8], select, outH[8], outL[8]);
45    mux2 m9(in0H[9], in0L[9], in1H[9], in1L[9], select, outH[9], outL[9]);
46    mux2 m10(in0H[10], in0L[10], in1H[10], in1L[10], select, outH[10], outL[10]);
47    mux2 m11(in0H[11], in0L[11], in1H[11], in1L[11], select, outH[11], outL[11]);
48    mux2 m12(in0H[12], in0L[12], in1H[12], in1L[12], select, outH[12], outL[12]);
49    mux2 m13(in0H[13], in0L[13], in1H[13], in1L[13], select, outH[13], outL[13]);
50    mux2 m14(in0H[14], in0L[14], in1H[14], in1L[14], select, outH[14], outL[14]);
51    mux2 m15(in0H[15], in0L[15], in1H[15], in1L[15], select, outH[15], outL[15]);
52    mux2 m16(in0H[16], in0L[16], in1H[16], in1L[16], select, outH[16], outL[16]);
53    mux2 m17(in0H[17], in0L[17], in1H[17], in1L[17], select, outH[17], outL[17]);
54    mux2 m18(in0H[18], in0L[18], in1H[18], in1L[18], select, outH[18], outL[18]);
55    mux2 m19(in0H[19], in0L[19], in1H[19], in1L[19], select, outH[19], outL[19]);
56    mux2 m20(in0H[20], in0L[20], in1H[20], in1L[20], select, outH[20], outL[20]);
57    mux2 m21(in0H[21], in0L[21], in1H[21], in1L[21], select, outH[21], outL[21]);
58    mux2 m22(in0H[22], in0L[22], in1H[22], in1L[22], select, outH[22], outL[22]);
59    mux2 m23(in0H[23], in0L[23], in1H[23], in1L[23], select, outH[23], outL[23]);
60    mux2 m24(in0H[24], in0L[24], in1H[24], in1L[24], select, outH[24], outL[24]);
61    mux2 m25(in0H[25], in0L[25], in1H[25], in1L[25], select, outH[25], outL[25]);
62    mux2 m26(in0H[26], in0L[26], in1H[26], in1L[26], select, outH[26], outL[26]);
63    mux2 m27(in0H[27], in0L[27], in1H[27], in1L[27], select, outH[27], outL[27]);
64    mux2 m28(in0H[28], in0L[28], in1H[28], in1L[28], select, outH[28], outL[28]);
65    mux2 m29(in0H[29], in0L[29], in1H[29], in1L[29], select, outH[29], outL[29]);
66    mux2 m30(in0H[30], in0L[30], in1H[30], in1L[30], select, outH[30], outL[30]);
67    mux2 m31(in0H[31], in0L[31], in1H[31], in1L[31], select, outH[31], outL[31]);
68 endmodule // mux2x32
```

```
69
70  module mux4x32(in00H, in00L, in01H, in01L, in10H, in10L, in11H, in11L,
71                 select, outH, outL);
72
73     input [31:0] in00H, in00L, in01H, in01L, in10H, in10L, in11H, in11L;
74     input [1:0]          select;
75     output [31:0] outH, outL;
76
77     wire [31:0]           w0H, w0L, w1H, w1L;
78
79     mux2x32 m1(in00H, in00L, in01H, in01L, select[0], w0H, w0L);
80     mux2x32 m2(in10H, in10L, in11H, in11L, select[0], w1H, w1L);
81     mux2x32 m3(w0H, w0L, w1H, w1L, select[1], outH, outL);
82  endmodule // mux4x32
83
84  module mux8x32(in000H, in000L, in001H, in001L, in010H, in010L,
85                 in011H, in011L, in100H, in100L, in101H, in101L,
86                 in110H, in110L, in111H, in111L, select, outH, outL);
87     input [31:0] in000H, in000L, in001H, in001L, in010H, in010L,
88                  in011H, in011L, in100H, in100L, in101H, in101L,
89                  in110H, in110L, in111H, in111L;
90     input [2:0]          select;
91     output [31:0] outH, outL;
92
93     wire [31:0]           w0H, w0L, w1H, w1L;
94
95     mux4x32 m1(in000H, in000L, in001H, in001L, in010H, in010L,
96                in011H, in011L, select[1:0], w0H, w0L);
97     mux4x32 m2(in100H, in100L, in101H, in101L, in110H, in110L,
98                in111H, in111L, select[1:0], w1H, w1L);
99     mux2x32 m3(w0H, w0L, w1H, w1L, select[2], outH, outL);
100 endmodule // mux8x32
101
102 module mux16x32(in0000H, in0000L, in0001H, in0001L, in0010H, in0010L,
103                 in0011H, in0011L, in0100H, in0100L, in0101H, in0101L,
104                 in0110H, in0110L, in0111H, in0111L, in1000H, in1000L,
105                 in1001H, in1001L, in1010H, in1010L, in1011H, in1011L,
106                 in1100H, in1100L, in1101H, in1101L, in1110H, in1110L,
107                 in1111H, in1111L, select, outH, outL);
108    input [31:0] in0000H, in0000L, in0001H, in0001L, in0010H, in0010L,
109                 in0011H, in0011L, in0100H, in0100L, in0101H, in0101L,
110                 in0110H, in0110L, in0111H, in0111L, in1000H, in1000L,
111                 in1001H, in1001L, in1010H, in1010L, in1011H, in1011L,
112                 in1100H, in1100L, in1101H, in1101L, in1110H, in1110L,
113                 in1111H, in1111L;
114    input [3:0]          select;
115    output [31:0] outH, outL;
116
117    wire [31:0]           w0H, w0L, w1H, w1L;
118
119    mux8x32 m1(in0000H, in0000L, in0001H, in0001L, in0010H, in0010L,
120                        in0011H, in0011L, in0100H, in0100L, in0101H, in0101L,
```

```
                      in0110H, in0110L, in0111H, in0111L, select[2:0], w0H, w0L);
   mux8x32 m2(in1000H, in1000L, in1001H, in1001L, in1010H, in1010L,
                      in1011H, in1011L, in1100H, in1100L, in1101H, in1101L,
                      in1110H, in1110L, in1111H, in1111L, select[2:0], w1H, w1L);
   mux2x32 m3(w0H, w0L, w1H, w1L, select[3], outH, outL);
endmodule // mux16x32

module mux32x32(in00000H, in00000L, in00001H, in00001L, in00010H, in00010L,
                in00011H, in00011L, in00100H, in00100L, in00101H, in00101L,
                in00110H, in00110L, in00111H, in00111L, in01000H, in01000L,
                in01001H, in01001L, in01010H, in01010L, in01011H, in01011L,
                in01100H, in01100L, in01101H, in01101L, in01110H, in01110L,
                in01111H, in01111L, in10000H, in10000L, in10001H, in10001L,
                in10010H, in10010L, in10011H, in10011L, in10100H, in10100L,
                in10101H, in10101L, in10110H, in10110L, in10111H, in10111L,
                in11000H, in11000L, in11001H, in11001L, in11010H, in11010L,
                in11011H, in11011L, in11100H, in11100L, in11101H, in11101L,
                in11110H, in11110L, in11111H, in11111L, select, outH, outL);
   input [31:0] in00000H, in00000L, in00001H, in00001L, in00010H, in00010L,
                in00011H, in00011L, in00100H, in00100L, in00101H, in00101L,
                in00110H, in00110L, in00111H, in00111L, in01000H, in01000L,
                in01001H, in01001L, in01010H, in01010L, in01011H, in01011L,
                in01100H, in01100L, in01101H, in01101L, in01110H, in01110L,
                in01111H, in01111L, in10000H, in10000L, in10001H, in10001L,
                in10010H, in10010L, in10011H, in10011L, in10100H, in10100L,
                in10101H, in10101L, in10110H, in10110L, in10111H, in10111L,
                in11000H, in11000L, in11001H, in11001L, in11010H, in11010L,
                in11011H, in11011L, in11100H, in11100L, in11101H, in11101L,
                in11110H, in11110L, in11111H, in11111L;
   input [4:0]         select;
   output [31:0] outH, outL;

   wire [31:0]          w0H, w0L, w1H, w1L;

   mux16x32 m1(in00000H, in00000L, in00001H, in00001L, in00010H, in00010L,
                       in00011H, in00011L, in00100H, in00100L, in00101H, in00101L,
                       in00110H, in00110L, in00111H, in00111L, in01000H, in01000L,
                       in01001H, in01001L, in01010H, in01010L, in01011H, in01011L,
                       in01100H, in01100L, in01101H, in01101L, in01110H, in01110L,
                       in01111H, in01111L, select[3:0], w0H, w0L);
   mux16x32 m2(in10000H, in10000L, in10001H, in10001L, in10010H, in10010L,
                       in10011H, in10011L, in10100H, in10100L, in10101H, in10101L,
                       in10110H, in10110L, in10111H, in10111L, in11000H, in11000L,
                       in11001H, in11001L, in11010H, in11010L, in11011H, in11011L,
                       in11100H, in11100L, in11101H, in11101L, in11110H, in11110L,
                       in11111H, in11111L, select[3:0], w1H, w1L);
   mux2x32 m3(w0H, w0L, w1H, w1L, select[4], outH, outL);
endmodule // mux32x32

module mux4(in0H, in0L, in1H, in1L, in2H, in2L, in3H, in3L,
            select, outH, outL);
   input in0H, in0L, in1H, in1L, in2H, in2L, in3H, in3L;
```

```
173     input [1:0] select;
174     output outH, outL;
175     wire   w01H, w01L, w23H, w23L;
176
177     mux2 m01(in0H, in0L, in1H, in1L, select[0], w01H, w01L);
178     mux2 m23(in2H, in2L, in3H, in3L, select[0], w23H, w23L);
179     mux2 mOUT(w01H, w01L, w23H, w23L, select[1], outH, outL);
180  endmodule // mux4


  1  // Register Modules
  2  // Asynchronous Processor
  3  // Robert Webb
  4
  5  module Dlatch(DH, DL, En, r, QH, QL);
  6     input DH, DL, En, r;
  7     output QH, QL;
  8
  9     wire   enDH,enDL, Hr, Lr, nr;
 10     wire   notnotQH, notLr;
 11
 12     and #2 (enDH, DH, En), (enDL, DL, En), (notnotQH, notLr, QH), (Hr, enDH, nr);
 13     not (notLr, Lr), (QL, notnotQH), (nr, r);
 14     or #1 (QH, Hr, notnotQH), (Lr, r, enDL);
 15  endmodule // Dlatch
 16
 17  module PosTrigger(clk, trig);
 18     input clk;
 19     output trig;
 20
 21     wire   notclk;
 22
 23     not #3  (notclk, clk);
 24     and  (trig, clk, notclk);
 25  endmodule // PosTrigger
 26
 27  module reg32(DH, DL, En, clk, r, QH, QL);
 28     input [31:0] DL, DH;
 29     input        En, clk, r;
 30     output [31:0] QH, QL;
 31
 32     wire         Enr, trig;
 33
 34     PosTrigger p(clk, trig);
 35     and #1 (Enr, En, trig);
 36     Dlatch r0(DH[0], DL[0], Enr, r, QH[0], QL[0]);
 37     Dlatch r1(DH[1], DL[1], Enr, r, QH[1], QL[1]);
 38     Dlatch r2(DH[2], DL[2], Enr, r, QH[2], QL[2]);
 39     Dlatch r3(DH[3], DL[3], Enr, r, QH[3], QL[3]);
 40     Dlatch r4(DH[4], DL[4], Enr, r, QH[4], QL[4]);
 41     Dlatch r5(DH[5], DL[5], Enr, r, QH[5], QL[5]);
 42     Dlatch r6(DH[6], DL[6], Enr, r, QH[6], QL[6]);
 43     Dlatch r7(DH[7], DL[7], Enr, r, QH[7], QL[7]);
```

```
44     Dlatch r8(DH[8], DL[8], Enr, r, QH[8], QL[8]);
45     Dlatch r9(DH[9], DL[9], Enr, r, QH[9], QL[9]);
46     Dlatch r10(DH[10], DL[10], Enr, r, QH[10], QL[10]);
47     Dlatch r11(DH[11], DL[11], Enr, r, QH[11], QL[11]);
48     Dlatch r12(DH[12], DL[12], Enr, r, QH[12], QL[12]);
49     Dlatch r13(DH[13], DL[13], Enr, r, QH[13], QL[13]);
50     Dlatch r14(DH[14], DL[14], Enr, r, QH[14], QL[14]);
51     Dlatch r15(DH[15], DL[15], Enr, r, QH[15], QL[15]);
52     Dlatch r16(DH[16], DL[16], Enr, r, QH[16], QL[16]);
53     Dlatch r17(DH[17], DL[17], Enr, r, QH[17], QL[17]);
54     Dlatch r18(DH[18], DL[18], Enr, r, QH[18], QL[18]);
55     Dlatch r19(DH[19], DL[19], Enr, r, QH[19], QL[19]);
56     Dlatch r20(DH[20], DL[20], Enr, r, QH[20], QL[20]);
57     Dlatch r21(DH[21], DL[21], Enr, r, QH[21], QL[21]);
58     Dlatch r22(DH[22], DL[22], Enr, r, QH[22], QL[22]);
59     Dlatch r23(DH[23], DL[23], Enr, r, QH[23], QL[23]);
60     Dlatch r24(DH[24], DL[24], Enr, r, QH[24], QL[24]);
61     Dlatch r25(DH[25], DL[25], Enr, r, QH[25], QL[25]);
62     Dlatch r26(DH[26], DL[26], Enr, r, QH[26], QL[26]);
63     Dlatch r27(DH[27], DL[27], Enr, r, QH[27], QL[27]);
64     Dlatch r28(DH[28], DL[28], Enr, r, QH[28], QL[28]);
65     Dlatch r29(DH[29], DL[29], Enr, r, QH[29], QL[29]);
66     Dlatch r30(DH[30], DL[30], Enr, r, QH[30], QL[30]);
67     Dlatch r31(DH[31], DL[31], Enr, r, QH[31], QL[31]);
68  endmodule // reg32
69
70  module registerfile(outreg1, outreg2, inreg, invH, invL,
71                      out1H, out1L, out2H, out2L, clk, clear, rst);
72     input [4:0] outreg1, outreg2, inreg;
73     input [31:0] invH, invL;
74     input         clk, clear, rst;
75     output [31:0] out1H, out1L, out2H, out2L;
76
77     wire [31:0]          e;
78     wire [31:0] r0H, r1H, r2H, r3H, r4H, r5H, r6H, r7H,
79                 r8H, r9H, r10H, r11H, r12H, r13H, r14H, r15H,
80                 r16H, r17H, r18H, r19H, r20H, r21H, r22H, r23H,
81                 r24H, r25H, r26H, r27H, r28H, r29H, r30H, r31H;
82     wire [31:0] r0L, r1L, r2L, r3L, r4L, r5L, r6L, r7L,
83                 r8L, r9L, r10L, r11L, r12L, r13L, r14L, r15L,
84                 r16L, r17L, r18L, r19L, r20L, r21L, r22L, r23L,
85                 r24L, r25L, r26L, r27L, r28L, r29L, r30L, r31L;
86
87     always @ (posedge clk)
88          if (rst) $strobe("Wrote %d to register $%d %d", invH, inreg, $time);
89
90     enable en(inreg, rst, e);
91     reg32 r0(32'b0, ~32'b0,
92              1, clk, clear, r0H, r0L);
93     reg32 r1(invH, invL, e[1], clk, clear, r1H, r1L);
94     reg32 r2(invH, invL, e[2], clk, clear, r2H, r2L);
95     reg32 r3(invH, invL, e[3], clk, clear, r3H, r3L);
```

```
96      reg32 r4(invH, invL, e[4], clk, clear, r4H, r4L);
97      reg32 r5(invH, invL, e[5], clk, clear, r5H, r5L);
98      reg32 r6(invH, invL, e[6], clk, clear, r6H, r6L);
99      reg32 r7(invH, invL, e[7], clk, clear, r7H, r7L);
100     reg32 r8(invH, invL, e[8], clk, clear, r8H, r8L);
101     reg32 r9(invH, invL, e[9], clk, clear, r9H, r9L);
102     reg32 r10(invH, invL, e[10], clk, clear, r10H, r10L);
103     reg32 r11(invH, invL, e[11], clk, clear, r11H, r11L);
104     reg32 r12(invH, invL, e[12], clk, clear, r12H, r12L);
105     reg32 r13(invH, invL, e[13], clk, clear, r13H, r13L);
106     reg32 r14(invH, invL, e[14], clk, clear, r14H, r14L);
107     reg32 r15(invH, invL, e[15], clk, clear, r15H, r15L);
108     reg32 r16(invH, invL, e[16], clk, clear, r16H, r16L);
109     reg32 r17(invH, invL, e[17], clk, clear, r17H, r17L);
110     reg32 r18(invH, invL, e[18], clk, clear, r18H, r18L);
111     reg32 r19(invH, invL, e[19], clk, clear, r19H, r19L);
112     reg32 r20(invH, invL, e[20], clk, clear, r20H, r20L);
113     reg32 r21(invH, invL, e[21], clk, clear, r21H, r21L);
114     reg32 r22(invH, invL, e[22], clk, clear, r22H, r22L);
115     reg32 r23(invH, invL, e[23], clk, clear, r23H, r23L);
116     reg32 r24(invH, invL, e[24], clk, clear, r24H, r24L);
117     reg32 r25(invH, invL, e[25], clk, clear, r25H, r25L);
118     reg32 r26(invH, invL, e[26], clk, clear, r26H, r26L);
119     reg32 r27(invH, invL, e[27], clk, clear, r27H, r27L);
120     reg32 r28(invH, invL, e[28], clk, clear, r28H, r28L);
121     reg32 r29(invH, invL, e[29], clk, clear, r29H, r29L);
122     reg32 r30(invH, invL, e[30], clk, clear, r30H, r30L);
123     reg32 r31(invH, invL, e[31], clk, clear, r31H, r31L);
124
125     mux32x32 m1(r0H, r0L, r1H, r1L, r2H, r2L, r3H, r3L,
126                 r4H, r4L, r5H, r5L, r6H, r6L, r7H, r7L,
127                 r8H, r8L, r9H, r9L, r10H, r10L,r11H, r11L,
128                 r12H, r12L, r13H, r13L, r14H, r14L, r15H, r15L,
129                 r16H, r16L, r17H, r17L, r18H, r18L, r19H, r19L,
130                 r20H, r20L, r21H, r21L, r22H, r22L, r23H, r23L,
131                 r24H, r24L, r25H, r25L, r26H, r26L, r27H, r27L,
132                 r28H, r28L, r29H, r29L, r30H, r30L, r31H, r31L,
133                 outreg1, out1H, out1L);
134
135     mux32x32 m2(r0H, r0L, r1H, r1L, r2H, r2L, r3H, r3L,
136                 r4H, r4L, r5H, r5L, r6H, r6L, r7H, r7L,
137                 r8H, r8L, r9H, r9L, r10H, r10L,r11H, r11L,
138                 r12H, r12L, r13H, r13L, r14H, r14L, r15H, r15L,
139                 r16H, r16L, r17H, r17L, r18H, r18L, r19H, r19L,
140                 r20H, r20L, r21H, r21L, r22H, r22L, r23H, r23L,
141                 r24H, r24L, r25H, r25L, r26H, r26L, r27H, r27L,
142                 r28H, r28L, r29H, r29L, r30H, r30L, r31H, r31L,
143                 outreg2, out2H, out2L);
144 endmodule // registerfile
```

```
1  // Instruction Fetch
2  // Asynchronous Processor
```

```
// Robert Webb

module instructionfetch(pc, clk, instructionH, instructionL);
   input [31:0] pc;
   input         clk;
   output [31:0] instructionH, instructionL;

   reg [31:0]          instructions[0:60];
   reg [31:0]          instructionH, instructionL;

   always @ (posedge clk) #1
     begin
        instructionH = instructions[pc]; instructionL = ~instructions[pc];
        if (pc > 56) $stop;
     end

   initial
     begin
     instructions[0] = 32'b0;
     instructionH = instructions[0]; instructionL = ~instructions[0];
     instructions[4] = 32'b00100000000001010000000000000001; // addi $5, $0, 1
     instructions[8] = 32'b00100000000001000000000000010100; // addi $4, $0 ,20
     instructions[12] = 32'b10101100000000000000000000000000;// sw $0, $0(0)
     instructions[16] = 32'b00100000000000010000000000000001;// addi $1, $0, 1
     instructions[20] = 32'b10101100000000010000000000000100;// sw $1, $0(4)
     //Loop
     instructions[24] = 32'b10001100000001100000000000000000;// lw $6, $0(0)
     instructions[28] = 32'b10001100000001110000000000000100;// lw $7, $0(4)
     instructions[32] = 32'b00000000111001100100000000100000;// add $8, $7, $6
     instructions[36] = 32'b10101100000001110000000000000000;// sw $7, $0(0)
     instructions[40] = 32'b10101100000001000000000000000100;// sw $8, $0(4)
     instructions[44] = 32'b00100000100001001111111111111111;// addi $4, $4, -1
     instructions[48] = 32'b00010100101001001111111111111001;// bne $4, $5,-7
     //EndLoop
     instructions[52] = 32'b00000001000000000001000000100000;// add $2,$8,$0
     instructions[56] = 32'b00001000000000000000000000001101;// j 52
     end // initial begin
endmodule // instructionfetch
```

```
// Memory Module
// Asynchronous Processor
// Robert Webb

module memoryfile(readdataH, readdataL, addressH, addressL,
                  writedataH, writedataL, memwrite, memread, get, set);
   input [31:0] addressH, addressL, writedataH, writedataL;
   input         get, set, memwrite, memread;
   output [31:0] readdataH, readdataL;

   reg [31:0]          readdataH = 0, readdataL = 0;
   reg [31:0]          sys[127:0];

```

```
14      always @ (posedge get)
15        begin
16           if (memread)
17             begin
18                readdataH = sys[addressH]; readdataL = ~sys[addressH];
19                $strobe("Read %d from ad. %d  %d \n", readdataH, addressH, $time);
20             end
21        end
22      always @ (posedge set)
23        begin
24           if (memwrite)
25             begin
26                sys[addressH] = writedataH;
27                $strobe("Wrote %d to ad. %d  %d \n", writedataH, addressH, $time);
28             end
29        end
30  endmodule // memoryfile
```

```
1   // Control Modules
2   // Asynchronous Processor
3   // Robert Webb
4
5   module control(inst, CLK, regDist, Branch, Jump, ALUop, ALUSrc, regWrite,
6                  MemWrite, MemRead, MemtoReg);
7      input[5:0] inst;
8      input      CLK;
9
10     output     regDist, Branch, ALUSrc, regWrite,
11                MemRead, MemWrite, MemtoReg, Jump;
12     output [1:0] ALUop;
13
14     reg     regDist, Branch, ALUSrc,regWrite,
15                MemRead, MemWrite, MemtoReg, Jump;
16     reg [1:0] ALUop;
17
18     initial
19       begin
20          regWrite = 0; regDist = 0; Branch = 0; Jump = 0;
21          ALUop = 2'b00; ALUSrc = 0;
22          MemRead = 0; MemWrite = 0; MemtoReg = 0;
23       end
24
25     always
26     begin #1
27     if(inst == 6'b0) //addi
28       begin
29          regWrite = 1; regDist = 1; Branch = 0; Jump = 0;
30          ALUop = 2'b10; ALUSrc = 0; //$strobe("R-Type Instruction");
31          MemRead = 0; MemWrite = 0; MemtoReg = 0;
32       end // if (inst == 6'b0)
33
34     if(inst == 6'b001000) //addi
```

```
35       begin
36          regWrite = 1; regDist = 0; Branch = 0; Jump = 0;
37          ALUop = 2'b00; ALUSrc = 1; //$strobe("addi");
38          MemWrite = 0; MemRead = 0; MemtoReg = 0;
39       end // if (inst == 6'b001000)
40
41      if(inst == 6'b000101) //bne
42       begin
43          regWrite = 0; regDist = 0; Branch = 1; Jump = 0;
44          ALUop = 2'b01; ALUSrc = 0; //$strobe("bne");
45          MemRead = 0; MemWrite = 0; MemtoReg = 0;
46       end
47
48     if(inst == 6'b000010) //j
49       begin
50          regWrite = 0; regDist = 0; Branch = 0; Jump = 1;
51          ALUop = 2'b01; ALUSrc = 0; //$strobe("jump");
52          MemWrite = 0; MemWrite = 0; MemtoReg = 0;
53       end
54
55     if(inst == 6'b100011) //lw
56       begin
57          regWrite = 1; regDist = 0; Branch = 0; Jump = 0;
58          ALUop = 2'b00; ALUSrc = 1; //$strobe("load word");
59          MemWrite = 0; MemRead = 1; MemtoReg = 1;
60       end
61
62     if(inst == 6'b101011) //sw
63       begin
64          regWrite = 0; regDist = 0; Branch = 0; Jump = 0;
65          ALUop = 2'b00; ALUSrc = 1; //$strobe("store word");
66          MemWrite = 1; MemRead = 0; MemtoReg = 0;
67       end
68      end // always begin
69  endmodule // control
70
71  module ALUcontrol(inst, ContOP, op);
72     input[5:0] inst;
73     input [1:0]     ContOP;
74     output [2:0] op;
75
76     reg [2:0]         op;
77     initial op = 0;
78
79     always
80       begin #1
81         if(ContOP == 2'b00) op = 3'b010;  // addi lw sw
82         if(ContOP == 2'b01) op = 3'b110;  // bne
83         if(ContOP == 2'b10)  //R-type instructions
84           begin
85              if(inst == 6'b100000) op = 3'b010;  //add
86              if(inst == 6'b100100) op = 3'b000;  //and
```

```
87              if(inst == 6'b100010) op = 3'b110;  //sub
88              if(inst == 6'b100101) op = 3'b001;  //or
89              if(inst == 6'b101010) op = 3'b111;  //slt
90          end // if (ContOP = 2'b10)
91      end // always begin
92  endmodule // ALUcontrol


1   // ALU Modules
2   // Asynchronous Processor
3   // Robert Webb
4
5   module adder1(aH, aL, bH, bL, cinH, coutH, resultH, resultL);
6      input aH, aL, bH, bL, cinH;
7      output resultH, resultL, coutH;
8      wire   w0H, w0L, w1H, w1L, w2H, w2L,cinL;
9      wire   a1H, a1L, a2H, a2L, a3H, a3L, a4H, a4L, a5H, a5L, a6H, a6L,
10            o7H, o7L, o8H, o8L;
11
12     not (cinL, cinH);
13     and2  a11(w0H, w0L, aH, aL, cinH, cinL),
14       a21(w1H, w1L, bH, bL, cinH, cinL),
15       a31(w2H, w2L, aH, aL, bH, bL);
16     or (coutH, w0H, w1H, w2H);
17     and2  a1(a1H, a1L, bL, bH, cinL, cinH),  // ~b*~c
18       a2(a2H, a2L, bH, bL, cinH, cinL),     // b*c
19       a3(a3H, a3L, bH, bL, cinL, cinH),     // b*~c
20       a4(a4H, a4L, bL, bH, cinH, cinL),     // ~b*c
21       a5(a5H, a5L, aH, aL, o7H, o7L),       // a * o7
22       a6(a6H, a6L, aL, aH, o8H, o8L);       // ~a * o8
23     or2 o7(o7H, o7L, a1H, a1L, a2H, a2L),   //  a1 + a2
24       o8(o8H, o8L, a3H, a3L, a4H, a4L),     //  a3 + a4
25       o9(resultH, resultL, a5H, a5L, a6H, a6L);  //  a5 + a6
26  endmodule // adder1
27
28  module alu1(aH, aL, bH, bL, cinH, lessH, binv, op, coutH, resultH, resultL);
29     input aH,aL,bH,bL,cinH,lessH,binv;
30     input [1:0]         op;
31     output      resultH, resultL, coutH;
32
33     wire      fbH, fbL, w0H, w0L, w1H, w1L, setH, setL, lessL;
34
35     not (lessL, lessH);
36     mux2 bmux(bH, bL, bL, bH, binv, fbH, fbL);
37     and2 a(w0H, w0L, aH, aL, fbH, fbL);
38     or2 o(w1H, w1L, aH, aL, fbH, fbL);
39     adder1 add(aH, aL, fbH, fbL, cinH, coutH, setH, setL);
40     mux4 mOUT(w0H, w0L, w1H, w1L, setH, setL, lessH, LessL, op,
41               resultH, resultL);
42  endmodule // alu1
43
44  module alu1end(aH, aL, bH, bL, cinH, lessH, binv, op, setH, resultH, resultL);
45     input aH,aL,bH,bL,cinH,lessH,binv;
```

```
46     input [1:0]         op;
47     output      resultH, resultL, setH;
48
49     wire        fbH, fbL, w0H, w0L, w1H, w1L, coutH, setL, lessL;
50
51     not (lessL, lessH);
52     mux2 bmux(bH, bL, bL, bH, binv, fbH, fbL);
53     and2 a(w0H, w0L, aH, aL, fbH, fbL);
54     or2 o(w1H, w1L, aH, aL, fbH, fbL);
55     adder1 add(aH, aL, fbH, fbL, cinH, coutH, setH, setL);
56     mux4 mOUT(w0H, w0L, w1H, w1L, setH, setL, lessH, LessL, op,
57               resultH, resultL);
58  endmodule // alu1End
59
60  module alu32(aH, aL, bH, bL, cinH, op, resultH, resultL, notzero);
61     input [31:0] aH, aL, bH, bL;
62     input [2:0]         op;
63     input         cinH;
64
65     output         notzero;
66     output [31:0] resultH, resultL;
67
68     wire [30:0] w;
69     wire        lessH, lessL;
70
71     alu1 a0(aH[0], aL[0], bH[0], bL[0], op[2], less,
72             op[2], op[1:0], w[0], resultH[0], resultL[0]);
73     alu1 a1(aH[1], aL[1], bH[1], bL[1], w[0], 1'b0,
74             op[2], op[1:0], w[1], resultH[1], resultL[1]);
75     alu1 a2(aH[2], aL[2], bH[2], bL[2], w[1], 1'b0,
76             op[2], op[1:0], w[2], resultH[2], resultL[2]);
77     alu1 a3(aH[3], aL[3], bH[3], bL[3], w[2], 1'b0,
78             op[2], op[1:0], w[3], resultH[3], resultL[3]);
79     alu1 a4(aH[4], aL[4], bH[4], bL[4], w[3], 1'b0,
80             op[2], op[1:0], w[4], resultH[4], resultL[4]);
81     alu1 a5(aH[5], aL[5], bH[5], bL[5], w[4], 1'b0,
82             op[2], op[1:0], w[5], resultH[5], resultL[5]);
83     alu1 a6(aH[6], aL[6], bH[6], bL[6], w[5], 1'b0,
84             op[2], op[1:0], w[6], resultH[6], resultL[6]);
85     alu1 a7(aH[7], aL[7], bH[7], bL[7], w[6], 1'b0,
86             op[2], op[1:0], w[7], resultH[7], resultL[7]);
87     alu1 a8(aH[8], aL[8], bH[8], bL[8], w[7], 1'b0,
88             op[2], op[1:0], w[8], resultH[8], resultL[8]);
89     alu1 a9(aH[9], aL[9], bH[9], bL[9], w[8], 1'b0,
90             op[2], op[1:0], w[9], resultH[9], resultL[9]);
91     alu1 a10(aH[10], aL[10], bH[10], bL[10], w[9], 1'b0,
92              op[2], op[1:0], w[10], resultH[10], resultL[10]);
93     alu1 a11(aH[11], aL[11], bH[11], bL[11], w[10], 1'b0,
94              op[2], op[1:0], w[11], resultH[11], resultL[11]);
95     alu1 a12(aH[12], aL[12], bH[12], bL[12], w[11], 1'b0,
96              op[2], op[1:0], w[12], resultH[12], resultL[12]);
97     alu1 a13(aH[13], aL[13], bH[13], bL[13], w[12], 1'b0,
```

```
            op[2], op[1:0], w[13], resultH[13], resultL[13]);
    alu1 a14(aH[14], aL[14], bH[14], bL[14], w[13], 1'b0,
            op[2], op[1:0], w[14], resultH[14], resultL[14]);
    alu1 a15(aH[15], aL[15], bH[15], bL[15], w[14], 1'b0,
            op[2], op[1:0], w[15], resultH[15], resultL[15]);
    alu1 a16(aH[16], aL[16], bH[16], bL[16], w[15], 1'b0,
            op[2], op[1:0], w[16], resultH[16], resultL[16]);
    alu1 a17(aH[17], aL[17], bH[17], bL[17], w[16], 1'b0,
            op[2], op[1:0], w[17], resultH[17], resultL[17]);
    alu1 a18(aH[18], aL[18], bH[18], bL[18], w[17], 1'b0,
            op[2], op[1:0], w[18], resultH[18], resultL[18]);
    alu1 a19(aH[19], aL[19], bH[19], bL[19], w[18], 1'b0,
            op[2], op[1:0], w[19], resultH[19], resultL[19]);
    alu1 a20(aH[20], aL[20], bH[20], bL[20], w[19], 1'b0,
            op[2], op[1:0], w[20], resultH[20], resultL[20]);
    alu1 a21(aH[21], aL[21], bH[21], bL[21], w[20], 1'b0,
            op[2], op[1:0], w[21], resultH[21], resultL[21]);
    alu1 a22(aH[22], aL[22], bH[22], bL[22], w[21], 1'b0,
            op[2], op[1:0], w[22], resultH[22], resultL[22]);
    alu1 a23(aH[23], aL[23], bH[23], bL[23], w[22], 1'b0,
            op[2], op[1:0], w[23], resultH[23], resultL[23]);
    alu1 a24(aH[24], aL[24], bH[24], bL[24], w[23], 1'b0,
            op[2], op[1:0], w[24], resultH[24], resultL[24]);
    alu1 a25(aH[25], aL[25], bH[25], bL[25], w[24], 1'b0,
            op[2], op[1:0], w[25], resultH[25], resultL[25]);
    alu1 a26(aH[26], aL[26], bH[26], bL[26], w[25], 1'b0,
            op[2], op[1:0], w[26], resultH[26], resultL[26]);
    alu1 a27(aH[27], aL[27], bH[27], bL[27], w[26], 1'b0,
            op[2], op[1:0], w[27], resultH[27], resultL[27]);
    alu1 a28(aH[28], aL[28], bH[28], bL[28], w[27], 1'b0,
            op[2], op[1:0], w[28], resultH[28], resultL[28]);
    alu1 a29(aH[29], aL[29], bH[29], bL[29], w[28], 1'b0,
            op[2], op[1:0], w[29], resultH[29], resultL[29]);
    alu1 a30(aH[30], aL[30], bH[30], bL[30], w[29], 1'b0,
            op[2], op[1:0], w[30], resultH[30], resultL[30]);
    alu1end a31(aH[31], aL[31], bH[30], bL[31], w[30], 1'b0,
                op[2], op[1:0], less, resultH[31], resultL[031]);

    or #1 (notzero, resultH[31], resultH[30], resultH[29], resultH[28],
           resultH[27], resultH[26], resultH[25], resultH[24], resultH[23],
           resultH[22], resultH[21], resultH[20], resultH[19], resultH[18],
           resultH[17], resultH[16], resultH[15], resultH[14], resultH[13],
           resultH[12], resultH[11], resultH[10], resultH[9], resultH[8],
           resultH[7], resultH[6], resultH[5], resultH[4], resultH[3],
           resultH[2], resultH[1], resultH[0]);
endmodule // alu32
```

```
// Completeness Indicator Module
// Asynchronous Processor
// Robert Webb

module ready(inH, inL, out);
```

```
    input [31:0] inH, inL;
    output         out;

    reg                 out;
    wire [31:0] r;
    wire good, really, structout;

    xor #1 (r[0], inH[0], inL[0]), (r[1], inH[1], inL[1]),
      (r[2], inH[2], inL[2]), (r[3], inH[3], inL[3]),
      (r[4], inH[4], inL[4]), (r[5], inH[5], inL[5]),
      (r[6], inH[6], inL[6]), (r[7], inH[7], inL[7]),
      (r[8], inH[8], inL[8]), (r[9], inH[9], inL[9]),
      (r[10], inH[10], inL[10]), (r[11], inH[11], inL[11]),
      (r[12], inH[12], inL[12]), (r[13], inH[13], inL[13]),
      (r[14], inH[14], inL[14]), (r[15], inH[15], inL[15]),
      (r[16], inH[16], inL[16]), (r[17], inH[17], inL[17]),
      (r[18], inH[18], inL[18]), (r[19], inH[19], inL[19]),
      (r[20], inH[20], inL[20]), (r[21], inH[21], inL[21]),
      (r[22], inH[22], inL[22]), (r[23], inH[23], inL[23]),
      (r[24], inH[24], inL[24]), (r[25], inH[25], inL[25]),
      (r[26], inH[26], inL[26]), (r[27], inH[27], inL[27]),
      (r[28], inH[28], inL[28]), (r[29], inH[29], inL[29]),
      (r[30], inH[30], inL[30]), (r[31], inH[31], inL[31]);
    and #2 (good, r[0], r[1], r[2], r[3], r[4], r[5], r[6], r[7], r[8],
            r[9], r[10], r[11], r[12], r[13], r[14], r[15], r[16], r[17],
            r[18], r[19], r[20], r[21], r[22], r[23], r[24], r[25], r[26],
            r[27], r[28], r[29], r[30], r[31]);
    buf #3 (really, good);
    and #2 (structout, really, good);
    always #1
      if(structout == 1) out = 1; else out = 0;
endmodule // ready


// Asynchronous Processor
// Robert Webb

module CPU(resultH, resultL, temp, rdy, pc);
    output [31:0] resultH, resultL, pc;
    output           rdy, temp;

    wire           ALUready, PCready;
    reg                  set;
    initial
      begin set = 1; #3 set = 0; //To set all of the flip-flops to 0
      end

    wire [31:0] PC, PCL, PCp4H, PCp4L, sgnExtIMH, sgnExtIML, sgnExtIMx4H,
                sgnExtIMx4L, branchPCH, branchPCL, nextPCH, nextPCL,
                data1H, data1L, data2H, data2L,
                immedH, immedL, finaloutH, finaloutL, instH, instL,
                JumpPCH, JumpPCL, finalNextPCH, finalNextPCL,
                memResultH, memResultL, ALUresultH, ALUresultL;
```

```
20      wire        RegWrite, isBranch, takeJump, notzero, takeBranch,
21                  ALUsrc, RegDist, memwrite, memread, memtoreg,
22                  REGready, MEMset, MEMready, Fready,
23                  nowhere1, nowhere2, InstReady, pcregReady, memready;
24      wire [1:0]  ALUop;
25      wire [2:0]  RealALUop;
26      wire [4:0]  WriteReg, nowhere5;
27
28      buf #2 (pcregREADY, rdy); // DELAY FOR PC
29      reg32 pcreg(finalNextPCH, finalNextPCL, 1'b1, rdy, set, PC, PCL);
30      alu32 addpc(PC, PCL, 32'd4, ~32'd4, 1'b0, 3'b010, PCp4H, PCp4L, nowhere1);
31      signextend16to32 se(instH[15:0], instL[15:0], sgnExtIMH, sgnExtIML);
32      shiftleft2 sl(sgnExtIMH, sgnExtIML, sgnExtIMx4H, sgnExtIMx4L);
33      alu32 bPC(PCp4H, PCp4L, sgnExtIMx4H, sgnExtIMx4L, 1'b0, 3'b010,
34                branchPCH, branchPCL, nowhere2);
35      and (takeBranch, isBranch, notzero);
36      mux2x32 PCchooser(PCp4H, PCp4L, branchPCH, branchPCL, takeBranch,
37                        nextPCH, nextPCL);
38      jumpPC jumpPCcalc(PC[31:28], ~PC[31:28], instH[25:0], instL[25:0],
39                        JumpPCH, JumpPCL);
40      mux2x32 JumpChooser(nextPCH, nextPCL, JumpPCH, JumpPCL, takeJump,
41                          finalNextPCH, finalNextPCL);
42      ready PCr(finalNextPCH, finalNextPCL, PCready);
43      buf #9 (InstReady, rdy); // DELAY FOR IF
44      instructionfetch InsF(PC, InstReady, instH, instL);
45      control controlU(instH[31:26], clk, RegDist, isBranch, takeJump, ALUop,
46                       ALUsrc, RegWrite, memwrite, memread, memtoreg);
47      ALUcontrol ALUcontrolI(instH[5:0], ALUop, RealALUop);
48      mux2x5 WriteChooser(instH[20:16], instL[20:16], instH[15:11], instL[15:11],
49                          RegDist, WriteReg, nowhere5);
50      buf #3 (writeREGready, rdy); // DELAY FOR REG
51      registerfile regfile(instH[25:21], instH[20:16], WriteReg,
52                           finaloutH, finaloutL,
53                           data1H, data1L, data2H, data2L,
54                           writeREGready, set, RegWrite);
55      mux2x32 dataChooser(data2H, data2L, sgnExtIMH, sgnExtIML, ALUsrc,
56                          immedH, immedL);
57      alu32 RealALU(data1H, data1L, immedH, immedL, 1'b0, RealALUop,
58                    ALUresultH, ALUresultL, notzero);
59      ready rdALU(ALUresultH, ALUresultL, ALUready);
60      buf #1 (memready, rdy); // DELAY FOR MEM
61      memoryfile mainMEM(memResultH, memResultL,
62                         ALUresultH, ALUresultL,
63                         data2H, data2L,
64                         memwrite, memread, ALUready, memready);
65
66      mux2x32 memORalu(ALUresultH, ALUresultL, memResultH, memResultL, memtoreg,
67                       finaloutH, finaloutL);
68
69      ready reFin(finaloutH, finaloutL, Fready);
70      and (rdy, Fready, PCready, ALUready);
71
```

```
72     assign resultH = finaloutH;
73     assign resultL = finaloutL;
74     assign pc = PC;
75     assign temp = ALUready;
76  endmodule // CPU
77
78  module main;
79     wire [31:0] pc, resultH, resultL, result;
80     wire       r, temp;
81     CPU SingleCycleAsync(resultH, resultL, temp, r, pc);
82     assign result = resultH;
83     initial
84          #8000 $stop;
85     initial
86        $monitor( "pc=%d result = %b r=%b temp=%b time=%d",
87                  pc, result, r, temp, $time);
88  endmodule // main
```

Appendix C

The Complete Pipelined Asynchronous Source Code

```
1  // Buffer Modules
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module enable(i, rst, enbit);
6     input [4:0] i;
7     input       rst;
8     output [31:0] enbit;
9
10    wire [4:0] ni;
11
12    not (ni[0], i[0]), (ni[1], i[1]), (ni[2], i[2]),
13      (ni[3], i[3]), (ni[4], i[4]);
14
15    and #1 (enbit[0], ni[4], ni[3], ni[2], ni[1], ni[0], rst),
16      (enbit[1], ni[4], ni[3], ni[2], ni[1], i[0], rst),
17      (enbit[2], ni[4], ni[3], ni[2], i[1], ni[0], rst),
18      (enbit[3], ni[4], ni[3], ni[2], i[1], i[0], rst),
19      (enbit[4], ni[4], ni[3], i[2], ni[1], ni[0], rst),
20      (enbit[5], ni[4], ni[3], i[2], ni[1], i[0], rst),
21      (enbit[6], ni[4], ni[3], i[2], i[1], ni[0], rst),
22      (enbit[7], ni[4], ni[3], i[2], i[1], i[0], rst),
23      (enbit[8], ni[4], i[3], ni[2], ni[1], ni[0], rst),
24      (enbit[9], ni[4], i[3], ni[2], ni[1], i[0], rst),
25      (enbit[10], ni[4], i[3], ni[2], i[1], ni[0], rst),
26      (enbit[11], ni[4], i[3], ni[2], i[1], i[0], rst),
27      (enbit[12], ni[4], i[3], i[2], ni[1], ni[0], rst),
28      (enbit[13], ni[4], i[3], i[2], ni[1], i[0], rst),
```

```
29      (enbit[14], ni[4], i[3], i[2], i[1], ni[0], rst),
30      (enbit[15], ni[4], i[3], i[2], i[1], i[0], rst),
31      (enbit[16], i[4], ni[3], ni[2], ni[1], ni[0], rst),
32      (enbit[17], i[4], ni[3], ni[2], ni[1], i[0], rst),
33      (enbit[18], i[4], ni[3], ni[2], i[1], ni[0], rst),
34      (enbit[19], i[4], ni[3], ni[2], i[1], i[0], rst),
35      (enbit[20], i[4], ni[3], i[2], ni[1], ni[0], rst),
36      (enbit[21], i[4], ni[3], i[2], ni[1], i[0], rst),
37      (enbit[22], i[4], ni[3], i[2], i[1], ni[0], rst),
38      (enbit[23], i[4], ni[3], i[2], i[1], i[0], rst),
39      (enbit[24], i[4], i[3], ni[2], ni[1], ni[0], rst),
40      (enbit[25], i[4], i[3], ni[2], ni[1], i[0], rst),
41      (enbit[26], i[4], i[3], ni[2], i[1], ni[0], rst),
42      (enbit[27], i[4], i[3], ni[2], i[1], i[0], rst),
43      (enbit[28], i[4], i[3], i[2], ni[1], ni[0], rst),
44      (enbit[29], i[4], i[3], i[2], ni[1], i[0], rst),
45      (enbit[30], i[4], i[3], i[2], i[1], ni[0], rst),
46      (enbit[31], i[4], i[3], i[2], i[1], i[0], rst);
47  endmodule // enable
48
49  module signextend16to32(imm, out);
50     input [15:0] imm;
51
52     output [31:0] out;
53
54     buf
55       (out[0], imm[0]), (out[1], imm[1]), (out[2], imm[2]), (out[3], imm[3]),
56       (out[4], imm[4]), (out[5], imm[5]), (out[6], imm[6]), (out[7], imm[7]),
57       (out[8], imm[8]), (out[9], imm[9]), (out[10], imm[10]), (out[11], imm[11]),
58       (out[12], imm[12]), (out[13], imm[13]), (out[14], imm[14]),
59       (out[15], imm[15]), (out[16], imm[15]), (out[17], imm[15]),
60       (out[18], imm[15]), (out[19], imm[15]), (out[20], imm[15]),
61       (out[21], imm[15]), (out[22], imm[15]), (out[23], imm[15]),
62       (out[24], imm[15]), (out[25], imm[15]), (out[26], imm[15]),
63       (out[27], imm[15]), (out[28], imm[15]), (out[29], imm[15]),
64       (out[30], imm[15]), (out[31], imm[15]);
65  endmodule // signextend16to32
66
67  module shiftleft2(inp, out);
68     input [31:0] inp;
69
70     output [31:0] out;
71
72     assign out[0] = 1'b0;
73     assign out[1] = 1'b0;
74
75     buf
76       (out[2], inp[0]), (out[3], inp[1]), (out[4], inp[2]), (out[5], inp[3]),
77       (out[6], inp[4]), (out[7], inp[5]), (out[8], inp[6]), (out[9], inp[7]),
78       (out[10], inp[8]), (out[11], inp[9]),  (out[12], inp[10]),
79       (out[13], inp[11]), (out[14], inp[12]), (out[15], inp[13]),
80       (out[16], inp[14]), (out[17], inp[15]), (out[18], inp[16]),
```

```
      (out[19], inp[17]), (out[20], inp[18]), (out[21], inp[19]),
      (out[22], inp[20]), (out[23], inp[21]), (out[24], inp[22]),
      (out[25], inp[23]), (out[26], inp[24]), (out[27], inp[25]),
      (out[28], inp[26]), (out[29], inp[27]), (out[30], inp[28]),
      (out[31], inp[29]);
endmodule // shiftleft2

module jumpPC(top, inp, out);

   input [3:0] top;
   input [25:0] inp;

   output [31:0] out;

   assign out[0] = 1'b0;
   assign out[1] = 1'b0;

   buf
     (out[2], inp[0]), (out[3], inp[1]), (out[4], inp[2]),  (out[5], inp[3]),
     (out[6], inp[4]), (out[7], inp[5]), (out[8], inp[6]),  (out[9], inp[7]),
     (out[10], inp[8]), (out[11], inp[9]), (out[12], inp[10]),
     (out[13], inp[11]), (out[14], inp[12]), (out[15], inp[13]),
     (out[16], inp[14]), (out[17], inp[15]), (out[18], inp[16]),
     (out[19], inp[17]), (out[20], inp[18]), (out[21], inp[19]),
     (out[22], inp[20]), (out[23], inp[21]), (out[24], inp[22]),
     (out[25], inp[23]), (out[26], inp[24]), (out[27], inp[25]),
     (out[28], top[0]), (out[29], top[1]),
     (out[30], top[2]), (out[31], top[3]);
endmodule // jumpPC
```

```
// C Elements
// Pipelined Asynchronous Processor
// Robert Webb

module Celement(a, b, c);
   input a, b;
   output c;

   reg           c = 0;

   always #1 if(a == 1 && b == 1) c = 1;
   always #1 if(a == 0 && b == 0) c = 0;
endmodule // Celement

module Celement3(a, b, c, d);
   input a, b, c;
   output d;

   reg           d = 0;

   always #1 if(a == 1 && b == 1 && c == 1) d = 1;
   always #1 if(a == 0 && b == 0 && c == 0) d = 0;
```

```
23 endmodule // Celement3
24
25 module CelementBUS(r, out);
26    input [31:0] r;
27    output         out;
28
29    reg                 out = 0;
30
31    always #1 if( r == 32'b11111111111111111111111111111111 ) out = 1;
32    always #1 if( r == 32'b0 ) out = 0;
33 endmodule // CelementBUS
```

```
1  // DIMS Asynchronous Gates
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module andD(outH, outL, AH, AL, BH, BL);
6     input AH, AL, BH, BL;
7     output outH, outL;
8
9     wire   LL, LH, HL, HH;
10
11    Celement ll(AL, BL, LL), lh(AL, BH, LH), hl(AH, BL, HL), hh(AH, BH, HH);
12    buf (outH, HH);
13    or #1 (outL, LL, LH, HL);
14 endmodule // andD
15
16 module orD(outH, outL, AH, AL, BH, BL);
17    input AH, AL, BH, BL;
18    output outH, outL;
19
20    wire   LL, LH, HL, HH;
21
22    Celement ll(AL, BL, LL), lh(AL, BH, LH), hl(AH, BL, HL), hh(AH, BH, HH);
23    buf (outL, LL);
24    or #1 (outH, HH, LH, HL);
25 endmodule // andD
```

```
1  // Traditional Mux Modules
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module mux(in0, in1, select, out);
6     input in0,in1,select;
7     output out;
8
9     wire   s0,w0,w1;
10
11    not #1 (s0, select);
12    and #1 (w0, s0, in0),
13           (w1, select, in1);
14    or #1 (out, w0, w1);
```

```
15 endmodule // mux
16
17 module mux5 (in0, in1, select, out);
18    input[4:0] in0,in1;
19    input      select;
20    output [4:0] out;
21
22    wire         s0;
23    wire [4:0]         w0,w1;
24
25        not #1 (s0, select);
26        and #1 (w0[0], s0, in0[0]),  (w0[1], s0, in0[1]),
27          (w0[2], s0, in0[2]), (w0[3], s0, in0[3]),
28          (w0[4], s0, in0[4]),  (w1[0], select, in1[0]),
29          (w1[1], select, in1[1]), (w1[2], select, in1[2]),
30          (w1[3], select, in1[3]), (w1[4], select, in1[4]);
31        or #1 (out[0], w0[0], w1[0]), (out[1], w0[1], w1[1]),
32          (out[2], w0[2], w1[2]), (out[3], w0[3], w1[3]),
33          (out[4], w0[4], w1[4]);
34 endmodule // mux5
35
36 module mux32 (in0, in1, select, out);
37    input [31:0] in0,in1;
38    input select;
39    output [31:0] out;
40
41    wire          s0;
42    wire [31:0] w0,w1;
43
44        not #1 (s0, select);
45        and #1
46          (w0[0], s0, in0[0]), (w0[1], s0, in0[1]), (w0[2], s0, in0[2]),
47          (w0[3], s0, in0[3]), (w0[4], s0, in0[4]), (w0[5], s0, in0[5]),
48          (w0[6], s0, in0[6]), (w0[7], s0, in0[7]), (w0[8], s0, in0[8]),
49          (w0[9], s0, in0[9]), (w0[10], s0, in0[10]), (w0[11], s0, in0[11]),
50          (w0[12], s0, in0[12]), (w0[13], s0, in0[13]), (w0[14], s0, in0[14]),
51          (w0[15], s0, in0[15]), (w0[16], s0, in0[16]), (w0[17], s0, in0[17]),
52          (w0[18], s0, in0[18]), (w0[19], s0, in0[19]), (w0[20], s0, in0[20]),
53          (w0[21], s0, in0[21]), (w0[22], s0, in0[22]), (w0[23], s0, in0[23]),
54          (w0[24], s0, in0[24]), (w0[25], s0, in0[25]), (w0[26], s0, in0[26]),
55          (w0[27], s0, in0[27]), (w0[28], s0, in0[28]), (w0[29], s0, in0[29]),
56          (w0[30], s0, in0[30]), (w0[31], s0, in0[31]),
57
58          (w1[0], select, in1[0]), (w1[1], select, in1[1]),
59          (w1[2], select, in1[2]), (w1[3], select, in1[3]),
60          (w1[4], select, in1[4]), (w1[5], select, in1[5]),
61          (w1[6], select, in1[6]), (w1[7], select, in1[7]),
62          (w1[8], select, in1[8]), (w1[9], select, in1[9]),
63          (w1[10], select, in1[10]), (w1[11], select, in1[11]),
64          (w1[12], select, in1[12]), (w1[13], select, in1[13]),
65          (w1[14], select, in1[14]), (w1[15], select, in1[15]),
66          (w1[16], select, in1[16]), (w1[17], select, in1[17]),
```

```
67          (w1[18], select, in1[18]), (w1[19], select, in1[19]),
68          (w1[20], select, in1[20]), (w1[21], select, in1[21]),
69          (w1[22], select, in1[22]), (w1[23], select, in1[23]),
70          (w1[24], select, in1[24]), (w1[25], select, in1[25]),
71          (w1[26], select, in1[26]), (w1[27], select, in1[27]),
72          (w1[28], select, in1[28]), (w1[29], select, in1[29]),
73          (w1[30], select, in1[30]), (w1[31], select, in1[31]);
74        or #1
75          (out[0],w1[0], w0[0]), (out[1],w1[1], w0[1]), (out[2],w1[2], w0[2]),
76          (out[3],w1[3], w0[3]), (out[4],w1[4], w0[4]), (out[5],w1[5], w0[5]),
77          (out[6],w1[6], w0[6]), (out[7],w1[7], w0[7]), (out[8],w1[8], w0[8]),
78          (out[9],w1[9], w0[9]), (out[10],w1[10], w0[10]),
79          (out[11],w1[11], w0[11]), (out[12],w1[12], w0[12]),
80          (out[13],w1[13], w0[13]), (out[14],w1[14], w0[14]),
81          (out[15],w1[15], w0[15]), (out[16],w1[16], w0[16]),
82          (out[17],w1[17], w0[17]), (out[18],w1[18], w0[18]),
83          (out[19],w1[19], w0[19]), (out[20],w1[20], w0[20]),
84          (out[21],w1[21], w0[21]), (out[22],w1[22], w0[22]),
85          (out[23],w1[23], w0[23]), (out[24],w1[24], w0[24]),
86          (out[25],w1[25], w0[25]), (out[26],w1[26], w0[26]),
87          (out[27],w1[27], w0[27]), (out[28],w1[28], w0[28]),
88          (out[29],w1[29], w0[29]), (out[30],w1[30], w0[30]),
89          (out[31],w1[31], w0[31]);
90  endmodule // mux32
91
92  module mux32x4(in00, in01, in10, in11, select, out);
93     input [31:0] in00, in01, in10, in11;
94     input [1:0]         select;
95     output [31:0] out;
96
97     wire [31:0]          w0, w1;
98
99     mux32 m1(in00, in01, select[0], w0);
100    mux32 m2(in10, in11, select[0], w1);
101    mux32 m3(w0, w1, select[1], out);
102 endmodule // mux32x4
103
104 module mux32x8(in000, in001, in010, in011, in100, in101, in110, in111,
105                select, out);
106    input [31:0] in000, in001, in010, in011, in100, in101, in110, in111;
107    input [2:0]         select;
108    output [31:0] out;
109
110    wire [31:0]          w0, w1;
111
112    mux32x4 m1(in000, in001, in010, in011, select[1:0], w0);
113    mux32x4 m2(in100, in101, in110, in111, select[1:0], w1);
114    mux32 m3(w0, w1, select[2], out);
115 endmodule // mux32
116
117 module mux32x16(in0000, in0001, in0010, in0011, in0100, in0101, in0110, in0111,
118                 in1000, in1001, in1010, in1011, in1100, in1101, in1110, in1111,
```

```
                select, out);
   input[31:0] in0000, in0001, in0010, in0011, in0100, in0101, in0110, in0111,
               in1000, in1001, in1010, in1011, in1100, in1101, in1110, in1111;
   input [3:0] select;
   output [31:0] out;

   wire [31:0]          w0, w1;

   mux32x8 m1(in0000, in0001, in0010, in0011, in0100, in0101, in0110, in0111,
              select[2:0], w0);
   mux32x8 m2(in1000, in1001, in1010, in1011, in1100, in1101, in1110, in1111,
              select[2:0], w1);
   mux32 m3(w0, w1, select[3], out);
endmodule // mux32x16

module mux32reg(regs00000, regs00001, regs00010, regs00011,
                regs00100, regs00101, regs00110, regs00111,
                regs01000, regs01001, regs01010, regs01011,
                regs01100, regs01101, regs01110, regs01111,
                regs10000, regs10001, regs10010, regs10011,
                regs10100, regs10101, regs10110, regs10111,
                regs11000, regs11001, regs11010, regs11011,
                regs11100, regs11101, regs11110, regs11111,
                select, out);

   input [31:0] regs00000, regs00001, regs00010, regs00011,
                regs00100, regs00101, regs00110, regs00111,
                regs01000, regs01001, regs01010, regs01011,
                regs01100, regs01101, regs01110, regs01111,
                regs10000, regs10001, regs10010, regs10011,
                regs10100, regs10101, regs10110, regs10111,
                regs11000, regs11001, regs11010, regs11011,
                regs11100, regs11101, regs11110, regs11111;
   input [4:0]         select;
   output [31:0] out;

   wire [31:0]          w0, w1;

   mux32x16 m1(regs00000, regs00001, regs00010, regs00011,
               regs00100, regs00101, regs00110, regs00111,
               regs01000, regs01001, regs01010, regs01011,
               regs01100, regs01101, regs01110, regs01111,
               select[3:0], w0);
   mux32x16 m2(regs10000, regs10001, regs10010, regs10011,
               regs10100, regs10101, regs10110, regs10111,
               regs11000, regs11001, regs11010, regs11011,
               regs11100, regs11101, regs11110, regs11111,
               select[3:0], w1);
   mux32 m3(w0, w1, select[4], out);
endmodule // mux32reg
```

```
// Conversion Modules
// Pipelined Asynchronous Processor
// Robert Webb

module b2dBIT(inp, req, outH, outL);
   input inp, req;
   output outH, outL;

   wire   inpL, inpH;

   not #1 (inpL, inp);
   buf #1 (inpH, inp); // So inpH and inpL transition simultaneously!

   and #1 (outH, inpH, req), (outL, inpL, req);
endmodule // bundled2dualBIT

module bundled2dual(inp, req, outH, outL);
   input req;
   input [31:0] inp;

   output [31:0] outH, outL;

   b2dBIT b0(inp[0], req, outH[0], outL[0]), b1(inp[1], req, outH[1], outL[1]),
   b2(inp[2], req, outH[2], outL[2]), b3(inp[3], req, outH[3], outL[3]),
   b4(inp[4], req, outH[4], outL[4]), b5(inp[5], req, outH[5], outL[5]),
   b6(inp[6], req, outH[6], outL[6]), b7(inp[7], req, outH[7], outL[7]),
   b8(inp[8], req, outH[8], outL[8]), b9(inp[9], req, outH[9], outL[9]),
   b10(inp[10], req, outH[10], outL[10]), b11(inp[11], req, outH[11], outL[11]),
   b12(inp[12], req, outH[12], outL[12]), b13(inp[13], req, outH[13], outL[13]),
   b14(inp[14], req, outH[14], outL[14]), b15(inp[15], req, outH[15], outL[15]),
   b16(inp[16], req, outH[16], outL[16]), b17(inp[17], req, outH[17], outL[17]),
   b18(inp[18], req, outH[18], outL[18]), b19(inp[19], req, outH[19], outL[19]),
   b20(inp[20], req, outH[20], outL[20]), b21(inp[21], req, outH[21], outL[21]),
   b22(inp[22], req, outH[22], outL[22]), b23(inp[23], req, outH[23], outL[23]),
   b24(inp[24], req, outH[24], outL[24]), b25(inp[25], req, outH[25], outL[25]),
   b26(inp[26], req, outH[26], outL[26]), b27(inp[27], req, outH[27], outL[27]),
   b28(inp[28], req, outH[28], outL[28]), b29(inp[29], req, outH[29], outL[29]),
   b30(inp[30], req, outH[30], outL[30]), b31(inp[31], req, outH[31], outL[31]);
endmodule // bundled2dual

module dual2bundled(inpH, inpL, out, req);
   input [31:0] inpH, inpL;

   output [31:0] out;
   output          req;

   wire [31:0]          r;

   assign out = inpH;

   or #1 (r[0], inpH[0], inpL[0]), (r[1], inpH[1], inpL[1]),
     (r[2], inpH[2], inpL[2]), (r[3], inpH[3], inpL[3]),
```

```
      (r[4], inpH[4], inpL[4]), (r[5], inpH[5], inpL[5]),
      (r[6], inpH[6], inpL[6]), (r[7], inpH[7], inpL[7]),
      (r[8], inpH[8], inpL[8]), (r[9], inpH[9], inpL[9]),
      (r[10], inpH[10], inpL[10]), (r[11], inpH[11], inpL[11]),
      (r[12], inpH[12], inpL[12]), (r[13], inpH[13], inpL[13]),
      (r[14], inpH[14], inpL[14]), (r[15], inpH[15], inpL[15]),
      (r[16], inpH[16], inpL[16]), (r[17], inpH[17], inpL[17]),
      (r[18], inpH[18], inpL[18]), (r[19], inpH[19], inpL[19]),
      (r[20], inpH[20], inpL[20]), (r[21], inpH[21], inpL[21]),
      (r[22], inpH[22], inpL[22]), (r[23], inpH[23], inpL[23]),
      (r[24], inpH[24], inpL[24]), (r[25], inpH[25], inpL[25]),
      (r[26], inpH[26], inpL[26]), (r[27], inpH[27], inpL[27]),
      (r[28], inpH[28], inpL[28]), (r[29], inpH[29], inpL[29]),
      (r[30], inpH[30], inpL[30]), (r[31], inpH[31], inpL[31]);
   CelementBUS dataCHECK(r, req);
endmodule // dual2bundled


// Dual-Rail Multiplexor Modules
// Pipelined Asynchronous Processor
// Robert Webb

module mux2DIMS(in0H, in0L, in1H, in1L, select, outH, outL);
   input in0H, in0L, in1H, in1L, select;
   output outH, outL;

   reg               outH = 0, outL = 0;

   always #2
     begin
        if (select == 0)
          begin
             outH = in0H; outL = in0L;
          end
        if (select == 1)
          begin
             outH = in1H; outL = in1L;
          end
     end
endmodule // mux2DIMS


module mux4DIMS(in0H, in0L, in1H, in1L, in2H, in2L, in3H, in3L,
            select, outH, outL);
   input in0H, in0L, in1H, in1L, in2H, in2L, in3H, in3L;
   input [1:0] select;
   output outH, outL;

   reg           outH = 0, outL = 0;

   always #2
     begin
        if (select == 2'b00)
```

```
36            begin
37               outH = in0H; outL = in0L;
38            end
39          if (select == 2'b01)
40            begin
41               outH = in1H; outL = in1L;
42            end
43          if (select == 2'b10)
44            begin
45               outH = in2H; outL = in2L;
46            end
47          if (select == 2'b11)
48            begin
49               outH = in3H; outL = in3L;
50            end
51       end
52  endmodule // mux4DIMS
```

```
1   // Control Modules
2   // Pipelined Asynchronous Processor
3   // Robert Webb
4
5   module jumpControl(inst, req, ack, jump);
6      input [5:0]  inst;
7      input         req, ack;
8      output          jump;
9
10     reg               jump;
11
12     initial
13       jump = 0;
14
15     always #1
16         if(inst == 6'b000010 && ack == 0)
17           #1 jump = 1;
18         else if(req == 0)
19           #1 jump = 0;
20  endmodule // jump
21
22  module control(inst, regDist, Branch, ALUop, ALUSrc, regWrite,
23                 MemWrite, MemRead, MemtoReg);
24     input[5:0]          inst;
25
26     output     regDist, Branch, ALUSrc, regWrite,
27                MemRead, MemWrite, MemtoReg;
28     output [1:0] ALUop;
29
30     reg     regDist, Branch, ALUSrc,regWrite,
31                MemRead, MemWrite, MemtoReg;
32     reg [1:0] ALUop;
33
34     initial
```

```
35        begin
36           regWrite = 0; regDist = 0; Branch = 0;
37           ALUop = 2'b00; ALUSrc = 0;
38           MemRead = 0; MemWrite = 0; MemtoReg = 0;
39        end
40
41      always
42      begin #1
43      if(inst == 6'b0) //addi
44        begin
45           regWrite = 1; regDist = 1; Branch = 0;
46           ALUop = 2'b10; ALUSrc = 0; //$strobe("R-Type (ADD) Instruction");
47           MemRead = 0; MemWrite = 0; MemtoReg = 0;
48        end // if (inst == 6'b0)
49
50      if(inst == 6'b001000) //addi
51        begin
52           regWrite = 1; regDist = 0; Branch = 0;
53           ALUop = 2'b00; ALUSrc = 1; //$strobe("addi");
54           MemWrite = 0; MemRead = 0; MemtoReg = 0;
55        end // if (inst == 6'b001000)
56
57       if(inst == 6'b000101) //bne
58        begin
59           regWrite = 0; regDist = 0; Branch = 1;
60           ALUop = 2'b01; ALUSrc = 0; //$strobe("bne");
61           MemRead = 0; MemWrite = 0; MemtoReg = 0;
62        end
63
64      if(inst == 6'b100011) //lw
65        begin
66           regWrite = 1; regDist = 0; Branch = 0;
67           ALUop = 2'b00; ALUSrc = 1; //$strobe("load word");
68           MemWrite = 0; MemRead = 1; MemtoReg = 1;
69        end
70
71      if(inst == 6'b101011) //sw
72        begin
73           regWrite = 0; regDist = 0; Branch = 0;
74           ALUop = 2'b00; ALUSrc = 1; //$strobe("store word");
75           MemWrite = 1; MemRead = 0; MemtoReg = 0;
76        end
77       end // always begin
78    endmodule // control
79
80    module ALUcontrol(inst, ContOP, op);
81      input[5:0] inst;
82      input [1:0]     ContOP;
83      output [2:0] op;
84
85      reg [2:0]         op;
86      initial op = 0;
```

```
87
88      always
89        begin #1
90          if(ContOP == 2'b00) op = 3'b010;  // addi lw sw
91          if(ContOP == 2'b01) op = 3'b000;  // bne (DOES NOTHING)
92          if(ContOP == 2'b10)  //R-type instructions
93            begin
94               if(inst == 6'b100000) op = 3'b010;  //add
95               if(inst == 6'b100100) op = 3'b000;  //and
96               if(inst == 6'b100010) op = 3'b110;  //sub
97               if(inst == 6'b100101) op = 3'b001;  //or
98               if(inst == 6'b101010) op = 3'b111;  //slt
99            end // if (ContOP = 2'b10)
100       end // always begin
101   endmodule // ALUcontrol


1   // Pipeline Modules
2   // Pipelined Asynchronous Processor
3   // Robert Webb
4
5   module mullergate(rIN, rOUT, aIN, aOUT, en);
6      input rIN, aIN;
7      output rOUT, aOUT, en;
8
9      not #1 (naIN, aIN);
10     Celement c1(naIN, rIN, en);
11     buf (aOUT, en);
12     buf #6 (rOUT, en);
13  endmodule // mullergate
14
15  module flushbranchdelayslot(inst, PCp4, PC, flush,
16                              inst_flushed, PCp4_flushed, PC_flushed);
17     input [31:0] inst, PCp4, PC;
18     input         flush;
19
20     output [31:0] inst_flushed, PCp4_flushed, PC_flushed;
21
22     reg [31:0]          inst_flushed, PCp4_flushed, PC_flushed;
23
24     always #1
25          if(flush == 0)
26            begin
27               inst_flushed = inst;
28               PCp4_flushed = PCp4;
29               PC_flushed = PC;
30            end
31          else
32            begin
33               inst_flushed = 0;
34               PCp4_flushed = 0;
35               PC_flushed = 0;
36            end
```

```
endmodule // flushbranchdelayslot

module noteq(in1, in2, neq);
   input [31:0] in1, in2;
   output         neq;

   wire [31:0]         neqbit;

   xor #1 (neqbit[0], in1[0], in2[0]), (neqbit[1], in1[1], in2[1]),
     (neqbit[2], in1[2], in2[2]), (neqbit[3], in1[3], in2[3]),
     (neqbit[4], in1[4], in2[4]), (neqbit[5], in1[5], in2[5]),
     (neqbit[6], in1[6], in2[6]), (neqbit[7], in1[7], in2[7]),
     (neqbit[8], in1[8], in2[8]), (neqbit[9], in1[9], in2[9]),
     (neqbit[10], in1[10], in2[10]), (neqbit[11], in1[11], in2[11]),
     (neqbit[12], in1[12], in2[12]), (neqbit[13], in1[13], in2[13]),
     (neqbit[14], in1[14], in2[14]), (neqbit[15], in1[15], in2[15]),
     (neqbit[16], in1[16], in2[16]), (neqbit[17], in1[17], in2[17]),
     (neqbit[18], in1[18], in2[18]), (neqbit[19], in1[19], in2[19]),
     (neqbit[20], in1[20], in2[20]), (neqbit[21], in1[21], in2[21]),
     (neqbit[22], in1[22], in2[22]), (neqbit[23], in1[23], in2[23]),
     (neqbit[24], in1[24], in2[24]), (neqbit[25], in1[25], in2[25]),
     (neqbit[26], in1[26], in2[26]), (neqbit[27], in1[27], in2[27]),
     (neqbit[28], in1[28], in2[28]), (neqbit[29], in1[29], in2[29]),
     (neqbit[30], in1[30], in2[30]), (neqbit[31], in1[31], in2[31]);

   or #1 (neq, neqbit[31], neqbit[30], neqbit[29], neqbit[28],
          neqbit[27], neqbit[26], neqbit[25], neqbit[24], neqbit[23],
          neqbit[22], neqbit[21], neqbit[20], neqbit[19], neqbit[18],
          neqbit[17], neqbit[16], neqbit[15], neqbit[14], neqbit[13],
          neqbit[12], neqbit[11], neqbit[10], neqbit[9], neqbit[8],
          neqbit[7], neqbit[6], neqbit[5], neqbit[4], neqbit[3],
          neqbit[2], neqbit[1], neqbit[0]);
endmodule // noteq


// Instruction Fetch
// Pipelined Asynchronous Processor
// Robert Webb

module instructionfetch(pc, ack, instruction, req);
   input [31:0] pc;
   input         ack;
   output [31:0] instruction;
   output          req;

   reg [31:0]          instructions[0:56];
   reg [31:0]          instruction;
   reg                  req;

   always @ (posedge ack)  //output has been used, so move on
     begin
        req = 0;
        instruction = instructions[pc];
```

```
19        end
20
21      always #1 // Stop if something goes amiss
22        if (pc > 56)
23          begin
24             $strobe("Went BEYOND END OF PROGRAM\n");
25             #1 $stop;
26          end
27
28      always #1 // If ack is 0, then ready for next request!
29        if (ack == 0)
30          #1 req = 1;
31
32      initial
33        begin
34        instructions[0] = 0;
35        instruction = instructions[0];
36        instructions[4] = 32'b00100000000001010000000000000001; // addi %5, %0, 1
37        instructions[8] = 32'b00100000000001000000000000010100; // addi $4,$0,20
38        instructions[12] = 32'b10101100000000000000000000000000; // sw $0, $0(0)
39        instructions[16] = 32'b00100000000000010000000000000001; //addi $1, $0, 1
40        instructions[20] = 32'b10101100000000010000000000000100; //sw $1, $0(4)
41        //Loop
42        instructions[24] = 32'b10001100000001100000000000000000; //lw $6, $0(0)
43        instructions[28] = 32'b10001100000001110000000000000100; //lw $7, $0(4)
44        instructions[32] = 32'b00000000111001100100000000100000;//add $8, $7, $6
45        instructions[36] = 32'b10101100000001110000000000000000; //sw $7, $0(0)
46        instructions[40] = 32'b10101100000010000000000000000100; //sw $8, $0(4)
47        instructions[44] = 32'b00100000100001001111111111111111; //addi $4, $4, -1
48        instructions[48] = 32'b00010100101001001111111111111001;// bne $4,$5,-7
49        //EndLoop
50        instructions[52] = 32'b00000001000000000001000000100000;// add $2,$8,$0
51        instructions[56] = 32'b00001000000000000000000000001101; // j 52
52        instructions[60] = 32'b0;
53        end // initial begin
54    endmodule // instructionfetch


1   // Register Modules
2   // Pipelined Asynchronous Processor
3   // Robert Webb
4
5   module Dlatch(Data, En, Q);
6      input Data, En;
7      output Q;
8
9      reg           D;
10
11     always #1
12       if (Data)  // IF Date = "Undefined" interpret as 0
13         D = Data;
14       else
15         D = 0;
```

```
16
17      wire   enD ,enNotD;
18      wire   notnotQ, notD;
19
20      not  (notD, D),
21        (notEnNotD, enNotD);
22      and #2 (enD, D, En),
23        (enNotD, notD, En);
24      or #1 (notnotQ, enD, Q);
25      and #2 (Q, notnotQ, notEnNotD);
26  endmodule // Dlatch
27
28  module reg1(D, En, clk, Q, set);
29      input D, En, clk, set;
30      output Q;
31
32      wire Enr, trig;
33
34      PosTrigger p(clk, trig);
35
36      and #1 (Enr, En, trig);
37
38      Dlatch r(D, Enr|set, Q);
39  endmodule // reg1
40
41  module reg3(D, En, clk, Q, set);
42      input [2:0] D;
43      input         En, clk, set;
44      output [2:0] Q;
45
46      wire          Enr, trig;
47
48      PosTrigger p(clk, trig);
49
50      and #1 (Enr, En, trig);
51
52      Dlatch r0(D[0], Enr|set, Q[0]); Dlatch r1(D[1], Enr|set, Q[1]);
53      Dlatch r2(D[2], Enr|set, Q[2]);
54  endmodule // reg3
55
56  module reg5(D, En, clk, Q, set);
57      input [4:0] D;
58      input       En, clk, set;
59      output [4:0] Q;
60
61      wire         Enr, trig;
62
63      PosTrigger p(clk, trig);
64
65      and #1 (Enr, En, trig);
66
67      Dlatch r0(D[0], Enr|set, Q[0]); Dlatch r1(D[1], Enr|set, Q[1]);
```

```
68     Dlatch r2(D[2], Enr|set, Q[2]); Dlatch r3(D[3], Enr|set, Q[3]);
69     Dlatch r4(D[4], Enr|set, Q[4]);
70  endmodule // reg5
71
72  module reg32(D, En, clk, Q, set);
73     input [31:0] D;
74     input        En, clk, set;
75     output [31:0] Q;
76
77     wire         Enr, trig;
78
79     PosTrigger p(clk, trig);
80
81     and #1 (Enr, En, trig);
82
83     Dlatch r0(D[0], Enr|set, Q[0]); Dlatch r1(D[1], Enr|set, Q[1]);
84     Dlatch r2(D[2], Enr|set, Q[2]); Dlatch r3(D[3], Enr|set, Q[3]);
85     Dlatch r4(D[4], Enr|set, Q[4]); Dlatch r5(D[5], Enr|set, Q[5]);
86     Dlatch r6(D[6], Enr|set, Q[6]); Dlatch r7(D[7], Enr|set, Q[7]);
87     Dlatch r8(D[8], Enr|set, Q[8]); Dlatch r9(D[9], Enr|set, Q[9]);
88     Dlatch r10(D[10], Enr|set, Q[10]); Dlatch r11(D[11], Enr|set, Q[11]);
89     Dlatch r12(D[12], Enr|set, Q[12]); Dlatch r13(D[13], Enr|set, Q[13]);
90     Dlatch r14(D[14], Enr|set, Q[14]); Dlatch r15(D[15], Enr|set, Q[15]);
91     Dlatch r16(D[16], Enr|set, Q[16]); Dlatch r17(D[17], Enr|set, Q[17]);
92     Dlatch r18(D[18], Enr|set, Q[18]); Dlatch r19(D[19], Enr|set, Q[19]);
93     Dlatch r20(D[20], Enr|set, Q[20]); Dlatch r21(D[21], Enr|set, Q[21]);
94     Dlatch r22(D[22], Enr|set, Q[22]); Dlatch r23(D[23], Enr|set, Q[23]);
95     Dlatch r24(D[24], Enr|set, Q[24]); Dlatch r25(D[25], Enr|set, Q[25]);
96     Dlatch r26(D[26], Enr|set, Q[26]); Dlatch r27(D[27], Enr|set, Q[27]);
97     Dlatch r28(D[28], Enr|set, Q[28]); Dlatch r29(D[29], Enr|set, Q[29]);
98     Dlatch r30(D[30], Enr|set, Q[30]); Dlatch r31(D[31], Enr|set, Q[31]);
99  endmodule // reg32
100
101 module PosTrigger(clk, trig);
102    input clk;
103    output trig;
104
105    wire   notclk;
106
107    not #3  (notclk, clk);
108    and  (trig, clk, notclk);
109 endmodule // PosTrigger
110
111 module registerfile(outreg1, outreg2, inreg, inv, out1, out2, clk, rst, set);
112    input [4:0] outreg1, outreg2, inreg;
113    input [31:0] inv;
114    input        clk, rst, set;
115    output [31:0] out1, out2;
116
117    wire [31:0]          e;
118    wire [31:0] r0, r1, r2, r3, r4, r5, r6, r7,
119                r8, r9, r10, r11, r12, r13, r14, r15,
```

```
                r16, r17, r18, r19, r20, r21, r22, r23,
                r24, r25, r26, r27, r28, r29, r30, r31;

    always @ (posedge clk)
      begin
         if (rst)
              begin
                    $strobe("Wrote %d to register $%d at time %d", inv, inreg, $time);
              end
      end

    enable en(inreg, rst, e);

    reg32 rg0(32'b0, 1, clk, r0, set); reg32 rg1(inv, e[1], clk, r1, set);
    reg32 rg2(inv, e[2], clk, r2, set); reg32 rg3(inv, e[3], clk, r3, set);
    reg32 rg4(inv, e[4], clk, r4, set); reg32 rg5(inv, e[5], clk, r5, set);
    reg32 rg6(inv, e[6], clk, r6, set); reg32 rg7(inv, e[7], clk, r7, set);
    reg32 rg8(inv, e[8], clk, r8, set); reg32 rg9(inv, e[9], clk, r9, set);
    reg32 rg10(inv, e[10], clk, r10, set); reg32 rg11(inv, e[11], clk, r11, set);
    reg32 rg12(inv, e[12], clk, r12, set); reg32 rg13(inv, e[13], clk, r13, set);
    reg32 rg14(inv, e[14], clk, r14, set); reg32 rg15(inv, e[15], clk, r15, set);
    reg32 rg16(inv, e[16], clk, r16, set); reg32 rg17(inv, e[17], clk, r17, set);
    reg32 rg18(inv, e[18], clk, r18, set); reg32 rg19(inv, e[19], clk, r19, set);
    reg32 rg20(inv, e[20], clk, r20, set); reg32 rg21(inv, e[21], clk, r21, set);
    reg32 rg22(inv, e[22], clk, r22, set); reg32 rg23(inv, e[23], clk, r23, set);
    reg32 rg24(inv, e[24], clk, r24, set); reg32 rg25(inv, e[25], clk, r25, set);
    reg32 rg26(inv, e[26], clk, r26, set); reg32 rg27(inv, e[27], clk, r27, set);
    reg32 rg28(inv, e[28], clk, r28, set); reg32 rg29(inv, e[29], clk, r29, set);
    reg32 rg30(inv, e[30], clk, r30, set); reg32 rg31(inv, e[31], clk, r31, set);

    mux32reg m1(r0, r1, r2, r3, r4, r5, r6, r7,
                r8, r9, r10, r11, r12, r13, r14, r15,
                r16, r17, r18, r19, r20, r21, r22, r23,
                r24, r25, r26, r27, r28, r29, r30, r31,
                outreg1, out1);

    mux32reg m2(r0, r1, r2, r3, r4, r5, r6, r7,
                r8, r9, r10, r11, r12, r13, r14, r15,
                r16, r17, r18, r19, r20, r21, r22, r23,
                r24, r25, r26, r27, r28, r29, r30, r31,
                outreg2, out2);
endmodule // registerfile
```

```
// Memory Module
// Pipelined Asynchronous Processor
// Robert Webb

module memory(readdata, address, writedata, memwrite, memread, req, ack);
    input [31:0] address, writedata;
    input         memread, memwrite, req;
    output [31:0] readdata;
    output          ack;
```

```
10
11      reg [31:0]          readdata = 0;
12      reg [31:0]          sys[127:0];
13      reg                  ack = 0;
14
15      always @ (posedge req)
16        begin
17           if (memread == 0 && memwrite == 0)
18             #1 ack = 1;
19           if (memread == 1)
20             begin
21                $strobe("Read  %d -- ad. %d at  %d", readdata, address, $time);
22                #3 readdata = sys[address]; // Delay is the memory delay
23                #1 ack = 1;
24             end
25           if (memwrite == 1)
26             begin
27                #1 sys[address] = writedata;
28                $strobe("Wrote %d to ad. %d at %d", writedata, address, $time);
29                ack = 1;
30             end
31        end
32      always @ (negedge req)
33        #1 ack = 0;
34
35  endmodule // memory
```

```
1   // DIMS 1-bit Adder
2   // Pipelined Asynchronous Processor
3   // Robert Webb
4
5   module adder1(aH, aL, bH, bL, cinH, cinL, coutH, coutL, resultH, resultL);
6      input aH, aL, bH, bL, cinH, cinL;
7      output resultH, resultL, coutH, coutL;
8
9      wire   LLL, LLH, LHL, LHH, HLL, HLH, HHL, HHH, gen, kill;
10
11     Celement3 lll(aL, bL, cinL, LLL);
12     Celement3 llh(aL, bL, cinH, LLH);
13     Celement3 lhl(aL, bH, cinL, LHL);
14     Celement3 lhh(aL, bH, cinH, LHH);
15     Celement3 hll(aH, bL, cinL, HLL);
16     Celement3 hlh(aH, bL, cinH, HLH);
17     Celement3 hhl(aH, bH, cinL, HHL);
18     Celement3 hhh(aH, bH, cinH, HHH);
19     Celement GEN(aH, bH, gen);
20     Celement KILL(aL, bL, kill);
21
22     or #1 (resultH, LLH, LHL, HLL, HHH),
23       (resultL, LLL, LHH, HLH, HHL),
24       (coutH, LHH, HLH, gen),
25       (coutL, LHL, HLL, kill);
```

```
26 endmodule // adder1


1  // ALU Modules
2  // Pipelined Asynchronous Processor
3  // Robert Webb
4
5  module alu1(aH, aL, bH, bL, cinH, cinL, lessH, lessL, binv, op,
6              coutH, coutL, resultH, resultL);
7     input aH,aL,bH,bL,cinH, cinL, lessH, lessL, binv;
8     input [1:0]          op;
9     output      resultH, resultL, coutH, coutL;
10
11    wire      fbH, fbL, w0H, w0L, w1H, w1L, setH, setL;
12
13    mux2DIMS bmux(bH, bL, bL, bH, binv, fbH, fbL);
14    andD a(w0H, w0L, aH, aL, fbH, fbL);
15    orD o(w1H, w1L, aH, aL, fbH, fbL);
16    adder1 add(aH, aL, fbH, fbL, cinH, cinL, coutH, coutL, setH, setL);
17    mux4DIMS mOUT(w0H, w0L, w1H, w1L, setH, setL, lessH, LessL, op,
18              resultH, resultL);
19 endmodule // alu1
20
21 module alu1end(aH, aL, bH, bL, cinH, cinL, lessH, lessL, binv, op,
22                setH, setL, resultH, resultL);
23    input aH,aL,bH,bL,cinH, cinL, lessH, lessL, binv;
24    input [1:0]          op;
25    output      resultH, resultL, setH, setL;
26
27    wire        fbH, fbL, w0H, w0L, w1H, w1L, coutH, coutL, setL, lessL;
28
29    mux2DIMS bmux(bH, bL, bL, bH, binv, fbH, fbL);
30    andD a(w0H, w0L, aH, aL, fbH, fbL);
31    orD o(w1H, w1L, aH, aL, fbH, fbL);
32    adder1 add(aH, aL, fbH, fbL, cinH, cinL, coutH, coutL, setH, setL);
33    mux4DIMS mOUT(w0H, w0L, w1H, w1L, setH, setL, lessH, LessL, op,
34              resultH, resultL);
35 endmodule // alu1End
36
37 module alu32(aH, aL, bH, bL, op, resultH, resultL, notzero);
38    input [31:0] aH, aL, bH, bL;
39    input [2:0]          op;
40
41    output         notzero;
42    output [31:0] resultH, resultL;
43
44    wire [30:0] wH, wL;
45    wire        lessH, lessL, aReady, bReady, inpReady, nOp2, cinH, cinL;
46
47    xor #1 (aReady, aH[0], aL[0]);
48    xor #1 (bReady, bH[0], bL[0]);
49    and #2 (inpReady, aReady, bReady);
50
```

```
51      not #1 (nOp2, op[2]);
52      and #2 (cinH, op[2], inpReady);
53      and #2 (cinL, nOp2, inpReady);
54
55      alu1 a0(aH[0], aL[0], bH[0], bL[0], cinH, cinL, lessH, lessL,
56              op[2], op[1:0], wH[0], wL[0], resultH[0], resultL[0]);
57      alu1 a1(aH[1], aL[1], bH[1], bL[1], wH[0], wL[0], 0, 1,
58              op[2], op[1:0], wH[1], wL[1], resultH[1], resultL[1]);
59      alu1 a2(aH[2], aL[2], bH[2], bL[2], wH[1], wL[1], 0, 1,
60              op[2], op[1:0], wH[2], wL[2], resultH[2], resultL[2]);
61      alu1 a3(aH[3], aL[3], bH[3], bL[3], wH[2], wL[2], 0, 1,
62              op[2], op[1:0], wH[3], wL[3], resultH[3], resultL[3]);
63      alu1 a4(aH[4], aL[4], bH[4], bL[4], wH[3], wL[3], 0, 1,
64              op[2], op[1:0], wH[4], wL[4], resultH[4], resultL[4]);
65      alu1 a5(aH[5], aL[5], bH[5], bL[5], wH[4], wL[4], 0, 1,
66              op[2], op[1:0], wH[5], wL[5], resultH[5], resultL[5]);
67      alu1 a6(aH[6], aL[6], bH[6], bL[6], wH[5], wL[5], 0, 1,
68              op[2], op[1:0], wH[6], wL[6], resultH[6], resultL[6]);
69      alu1 a7(aH[7], aL[7], bH[7], bL[7], wH[6], wL[6], 0, 1,
70              op[2], op[1:0], wH[7], wL[7], resultH[7], resultL[7]);
71      alu1 a8(aH[8], aL[8], bH[8], bL[8], wH[7], wL[7], 0, 1,
72              op[2], op[1:0], wH[8], wL[8], resultH[8], resultL[8]);
73      alu1 a9(aH[9], aL[9], bH[9], bL[9], wH[8], wL[8], 0, 1,
74              op[2], op[1:0], wH[9], wL[9], resultH[9], resultL[9]);
75      alu1 a10(aH[10], aL[10], bH[10], bL[10], wH[9], wL[9], 0, 1,
76               op[2], op[1:0], wH[10], wL[10], resultH[10], resultL[10]);
77      alu1 a11(aH[11], aL[11], bH[11], bL[11], wH[10], wL[10], 0, 1,
78               op[2], op[1:0], wH[11], wL[11], resultH[11], resultL[11]);
79      alu1 a12(aH[12], aL[12], bH[12], bL[12], wH[11], wL[11], 0, 1,
80               op[2], op[1:0], wH[12], wL[12], resultH[12], resultL[12]);
81      alu1 a13(aH[13], aL[13], bH[13], bL[13], wH[12], wL[12], 0, 1,
82               op[2], op[1:0], wH[13], wL[13], resultH[13], resultL[13]);
83      alu1 a14(aH[14], aL[14], bH[14], bL[14], wH[13], wL[13], 0, 1,
84               op[2], op[1:0], wH[14], wL[14], resultH[14], resultL[14]);
85      alu1 a15(aH[15], aL[15], bH[15], bL[15], wH[14], wL[14], 0, 1,
86               op[2], op[1:0], wH[15], wL[15], resultH[15], resultL[15]);
87      alu1 a16(aH[16], aL[16], bH[16], bL[16], wH[15], wL[15], 0, 1,
88               op[2], op[1:0], wH[16], wL[16], resultH[16], resultL[16]);
89      alu1 a17(aH[17], aL[17], bH[17], bL[17], wH[16], wL[16], 0, 1,
90               op[2], op[1:0], wH[17], wL[17], resultH[17], resultL[17]);
91      alu1 a18(aH[18], aL[18], bH[18], bL[18], wH[17], wL[17], 0, 1,
92               op[2], op[1:0], wH[18], wL[18], resultH[18], resultL[18]);
93      alu1 a19(aH[19], aL[19], bH[19], bL[19], wH[18], wL[18], 0, 1,
94               op[2], op[1:0], wH[19], wL[19], resultH[19], resultL[19]);
95      alu1 a20(aH[20], aL[20], bH[20], bL[20], wH[19], wL[19], 0, 1,
96               op[2], op[1:0], wH[20], wL[20], resultH[20], resultL[20]);
97      alu1 a21(aH[21], aL[21], bH[21], bL[21], wH[20], wL[20], 0, 1,
98               op[2], op[1:0], wH[21], wL[21], resultH[21], resultL[21]);
99      alu1 a22(aH[22], aL[22], bH[22], bL[22], wH[21], wL[21], 0, 1,
100              op[2], op[1:0], wH[22], wL[22], resultH[22], resultL[22]);
101     alu1 a23(aH[23], aL[23], bH[23], bL[23], wH[22], wL[22], 0, 1,
102              op[2], op[1:0], wH[23], wL[23], resultH[23], resultL[23]);
```

```
103     alu1 a24(aH[24], aL[24], bH[24], bL[24], wH[23], wL[23], 0, 1,
104              op[2], op[1:0], wH[24], wL[24], resultH[24], resultL[24]);
105     alu1 a25(aH[25], aL[25], bH[25], bL[25], wH[24], wL[24], 0, 1,
106              op[2], op[1:0], wH[25], wL[25], resultH[25], resultL[25]);
107     alu1 a26(aH[26], aL[26], bH[26], bL[26], wH[25], wL[25], 0, 1,
108              op[2], op[1:0], wH[26], wL[26], resultH[26], resultL[26]);
109     alu1 a27(aH[27], aL[27], bH[27], bL[27], wH[26], wL[26], 0, 1,
110              op[2], op[1:0], wH[27], wL[27], resultH[27], resultL[27]);
111     alu1 a28(aH[28], aL[28], bH[28], bL[28], wH[27], wL[27], 0, 1,
112              op[2], op[1:0], wH[28], wL[28], resultH[28], resultL[28]);
113     alu1 a29(aH[29], aL[29], bH[29], bL[29], wH[28], wL[28], 0, 1,
114              op[2], op[1:0], wH[29], wL[29], resultH[29], resultL[29]);
115     alu1 a30(aH[30], aL[30], bH[30], bL[30], wH[29], wL[29], 0, 1,
116              op[2], op[1:0], wH[30], wL[30], resultH[30], resultL[30]);
117     alu1end a31(aH[31], aL[31], bH[31], bL[31], wH[30], wL[30], 0, 1,
118                 op[2], op[1:0], lessH, lessL, resultH[31], resultL[31]);
119
120     or #1 (notZero, resultH[31], resultH[30], resultH[29], resultH[28],
121            resultH[27], resultH[26], resultH[25], resultH[24], resultH[23],
122            resultH[22], resultH[21], resultH[20], resultH[19], resultH[18],
123            resultH[17], resultH[16], resultH[15], resultH[14], resultH[13],
124            resultH[12], resultH[11], resultH[10], resultH[9], resultH[8],
125            resultH[7], resultH[6], resultH[5], resultH[4], resultH[3],
126            resultH[2], resultH[1], resultH[0]);
127 endmodule // alu32
128
129 module adder32(aH, aL, bH, bL, resultH, resultL);
130    input [31:0] aH, aL, bH, bL;
131
132    output [31:0] resultH, resultL;
133
134    wire [30:0] wH, wL;
135    wire        aReady, bReady, inpReady;
136
137    xor #1 (aReady, aH[0], aL[0]);
138    xor #1 (bReady, bH[0], bL[0]);
139    and #1 (inpReady, aReady, bReady);
140
141    adder1 a0(aH[0], aL[0], bH[0], bL[0], 0, inpReady, wH[0], wL[0],
142              resultH[0], resultL[0]);
143    adder1 a1(aH[1], aL[1], bH[1], bL[1], wH[0], wL[0], wH[1], wL[1],
144              resultH[1], resultL[1]);
145    adder1 a2(aH[2], aL[2], bH[2], bL[2], wH[1], wL[1], wH[2], wL[2],
146              resultH[2], resultL[2]);
147    adder1 a3(aH[3], aL[3], bH[3], bL[3], wH[2], wL[2], wH[3], wL[3],
148              resultH[3], resultL[3]);
149    adder1 a4(aH[4], aL[4], bH[4], bL[4], wH[3], wL[3], wH[4], wL[4],
150              resultH[4], resultL[4]);
151    adder1 a5(aH[5], aL[5], bH[5], bL[5], wH[4], wL[4], wH[5], wL[5],
152              resultH[5], resultL[5]);
153    adder1 a6(aH[6], aL[6], bH[6], bL[6], wH[5], wL[5], wH[6], wL[6],
154              resultH[6], resultL[6]);
```

```
155     adder1 a7(aH[7], aL[7], bH[7], bL[7], wH[6], wL[6], wH[7], wL[7],
156               resultH[7], resultL[7]);
157     adder1 a8(aH[8], aL[8], bH[8], bL[8], wH[7], wL[7], wH[8], wL[8],
158               resultH[8], resultL[8]);
159     adder1 a9(aH[9], aL[9], bH[9], bL[9], wH[8], wL[8], wH[9], wL[9],
160               resultH[9], resultL[9]);
161     adder1 a10(aH[10], aL[10], bH[10], bL[10], wH[9], wL[9], wH[10], wL[10],
162                resultH[10], resultL[10]);
163     adder1 a11(aH[11], aL[11], bH[11], bL[11], wH[10], wL[10], wH[11], wL[11],
164                resultH[11], resultL[11]);
165     adder1 a12(aH[12], aL[12], bH[12], bL[12], wH[11], wL[11], wH[12], wL[12],
166                resultH[12], resultL[12]);
167     adder1 a13(aH[13], aL[13], bH[13], bL[13], wH[12], wL[12], wH[13], wL[13],
168                resultH[13], resultL[13]);
169     adder1 a14(aH[14], aL[14], bH[14], bL[14], wH[13], wL[13], wH[14], wL[14],
170                resultH[14], resultL[14]);
171     adder1 a15(aH[15], aL[15], bH[15], bL[15], wH[14], wL[14], wH[15], wL[15],
172                resultH[15], resultL[15]);
173     adder1 a16(aH[16], aL[16], bH[16], bL[16], wH[15], wL[15], wH[16], wL[16],
174                resultH[16], resultL[16]);
175     adder1 a17(aH[17], aL[17], bH[17], bL[17], wH[16], wL[16], wH[17], wL[17],
176                resultH[17], resultL[17]);
177     adder1 a18(aH[18], aL[18], bH[18], bL[18], wH[17], wL[17], wH[18], wL[18],
178                resultH[18], resultL[18]);
179     adder1 a19(aH[19], aL[19], bH[19], bL[19], wH[18], wL[18], wH[19], wL[19],
180                resultH[19], resultL[19]);
181     adder1 a20(aH[20], aL[20], bH[20], bL[20], wH[19], wL[19], wH[20], wL[20],
182                resultH[20], resultL[20]);
183     adder1 a21(aH[21], aL[21], bH[21], bL[21], wH[20], wL[20], wH[21], wL[21],
184                resultH[21], resultL[21]);
185     adder1 a22(aH[22], aL[22], bH[22], bL[22], wH[21], wL[21], wH[22], wL[22],
186                resultH[22], resultL[22]);
187     adder1 a23(aH[23], aL[23], bH[23], bL[23], wH[22], wL[22], wH[23], wL[23],
188                resultH[23], resultL[23]);
189     adder1 a24(aH[24], aL[24], bH[24], bL[24], wH[23], wL[23], wH[24], wL[24],
190                resultH[24], resultL[24]);
191     adder1 a25(aH[25], aL[25], bH[25], bL[25], wH[24], wL[24], wH[25], wL[25],
192                resultH[25], resultL[25]);
193     adder1 a26(aH[26], aL[26], bH[26], bL[26], wH[25], wL[25], wH[26], wL[26],
194                resultH[26], resultL[26]);
195     adder1 a27(aH[27], aL[27], bH[27], bL[27], wH[26], wL[26], wH[27], wL[27],
196                resultH[27], resultL[27]);
197     adder1 a28(aH[28], aL[28], bH[28], bL[28], wH[27], wL[27], wH[28], wL[28],
198                resultH[28], resultL[28]);
199     adder1 a29(aH[29], aL[29], bH[29], bL[29], wH[28], wL[28], wH[29], wL[29],
200                resultH[29], resultL[29]);
201     adder1 a30(aH[30], aL[30], bH[30], bL[30], wH[29], wL[29], wH[30], wL[30],
202                resultH[30], resultL[30]);
203     adder1 a31(aH[31], aL[31], bH[31], bL[31], wH[30], wL[30],
204                nowhereH, nowhereL, resultH[31], resultL[31]);
205 endmodule // adder32
206
```

```
// Pipelined Asynchronous Processor
// Robert Webb

module CPU(pc, PC, PC_ID, PC_EX, s1out, s2in, s2out, s3in, s3out, temp);
   output [31:0] pc, PC, PC_ID, PC_EX, s1out, s2in, s2out, s3in, s3out;
   output          temp;

   reg                  set;

   initial
     begin set = 1; #3 set = 0; end //ZERO OUT ALL FLIP-FLOPS

   //IF
   wire [31:0] PCH, PCL, fourH, fourL, PCp4, PCp4H, PCp4L,
               PCp4_flushed, jPC, PCp4_or_jPC, nextPC,
               inst, inst_flushed, PC_flushed;
   wire        ack_to_IF, ack_to_IForJump, addstart,
               ifreq, addreq, if_add_req, jump,
               jumpUpdate, waitforPC, req_from_IF,
               latchIF;

   //ID
   wire [31:0] instPipe, PCp4pipe, PCp4pipeH, PCp4pipeL, data1, data2,
               imm, branchOff, branchOffH, branchOffL,
               branchPC, branchPCH, branchPCL, branchPC_to_IF;
   wire [1:0]  ALUop;
   wire [2:0]  RealALUop;
   wire [4:0]  writeReg;
   wire        req_to_ID, ack_from_ID, regDist, Branch, takeBranch, ALUsrc,
               regwrite, memwrite, memread, memtoreg, neq, ack_to_ID, bPCreq,
               regreq, req_from_ID, latchID, write_negedge;

   //EX
   wire [31:0] data1pipe, data2pipe, immPipe,
               ALUinput2, ALUinput1H, ALUinput1L, ALUinput2H, ALUinput2L,
               ALUresult, ALUresultH, ALUresultL,
               MEMresult, writeD;
   wire [4:0]  writeRegPipe;
   wire [2:0]  realALUopPipe;
   wire        req_to_EX, ack_from_EX, req_to_ALU, ALUreq, MEMreq,
               regwritePipe, memwritePipe, memreadPipe,
               memtoregPipe, ALUsrcPipe;

   reg32 pcreg(nextPC, 1, ack_to_IForJump, PC, set);
   not #1 (addstart, ack_to_IForJump);
   bundled2dual aluinp1(PC, addstart, PCH, PCL);
   bundled2dual aluinp2(32'd4, addstart, fourH, fourL);
   adder32 pcpfour(PCH, PCL, fourH, fourL, PCp4H, PCp4L);
   dual2bundled npc(PCp4H, PCp4L, PCp4, addreq);
   buf #6 (waitforPC, ack_to_IForJump); // DELAY FOR PC REGISTER
   instructionfetch instF(PC, waitforPC, inst, ifreq);
   jumpControl jc(inst[31:26], addreq | ifreq, waitforPC, jump);
```

```
53     jumpPC jpc(PC[31:28], inst[25:0], jPC);
54     mux32 pcChooser1(PCp4, jPC, jump, PCp4_or_jPC);
55     mux32 pcChooser2(PCp4_or_jPC, branchPC_to_IF, takeBranch, nextPC);
56     buf #6 (jumpUpdate, jump); // DELAY FOR NEXT PC
57     or #1 (ack_to_IForJump, jumpUpdate, ack_to_IF);
58     Celement combine_IF_ALU(ifreq, addreq, if_add_req);
59     and #1 (req_from_IF, if_add_req, ~jump);
60     flushbranchdelayslot f(inst, PCp4, PC, takeBranch,
61                            inst_flushed, PCp4_flushed, PC_flushed);
62  //************************************************************************
63     mullergate if_id(req_from_IF, req_to_ID, ack_from_ID, ack_to_IF, latchIF);
64     reg32 if_id_reg_inst(inst_flushed, 1, latchIF, instPipe, set);
65     reg32 if_id_reg_PCp4(PCp4_flushed, 1, latchIF, PCp4pipe, set);
66     reg32 if_id_reg_PC(PC_flushed, 1, latchIF, PC_ID, set);
67  //************************************************************************
68
69     control cunit(instPipe[31:26], regDist, Branch, ALUop, ALUsrc, regwrite,
70                   memwrite, memread, memtoreg);
71     ALUcontrol acunit(instPipe[5:0], ALUop, RealALUop);
72     registerfile rf(instPipe[25:21], instPipe[20:16], writeRegPipe, writeD,
73                     data1, data2, ack_from_EX, regwritePipe, set);
74     signextend16to32 i(instPipe[15:0], imm);
75     noteq n(data1, data2, neq);
76     mux5 writechooser(instPipe[20:16], instPipe[15:11], regDist, writeReg);
77     shiftleft2 bOFF(imm, branchOff);
78     bundled2dual bo2(branchOff, req_to_ID, branchOffH, branchOffL);
79     bundled2dual PCp4_2(PCp4pipe, req_to_ID, PCp4pipeH, PCp4pipeL);
80     adder32 branchPCcalc(branchOffH, branchOffL, PCp4pipeH, PCp4pipeL,
81                          branchPCH, branchPCL);
82     dual2bundled bPC(branchPCH, branchPCL, branchPC, bPCreq);
83     reg32 branchPCsaver(branchPC, 1, ~req_to_ID, branchPC_to_IF, set);
84     and #1 (takeBranch, neq, Branch);
85     buf #7 (regreq, req_to_ID); // DELAY FOR REG
86     not #1 (donttakebranch, takeBranch);
87     and #1 (dontneed, donttakebranch, regreq);
88     or #1(bPCreq_or_dontneed, dontneed);
89     Celement IDreq(regreq, bPCreq_or_dontneed, req_from_ID);
90     buf (ack_from_ID, ack_to_ID);
91
92  //************************************************************************
93     mullergate id_ex(req_from_ID, req_to_EX, ack_from_EX, ack_to_ID, latchID);
94     reg32 id_ex_reg_data1(data1, 1, latchID, data1pipe, set);
95     reg32 id_ex_reg_data2(data2, 1, latchID, data2pipe, set);
96     reg32 id_ex_reg_imm(imm, 1, latchID, immPipe, set);
97     reg5 id_ex_reg_writeReg(writeReg, 1, latchID, writeRegPipe, set);
98     reg3 id_ex_reg_RealALUop(RealALUop, 1, latchID, realALUopPipe, set);
99     reg1 id_ex_reg_regwrite(regwrite, 1, latchID, regwritePipe, set);
100    reg1 id_ex_reg_memwrite(memwrite, 1, latchID, memwritePipe, set);
101    reg1 id_ex_reg_memread(memread, 1, latchID, memreadPipe, set);
102    reg1 id_ex_reg_memtoreg(memtoreg, 1, latchID, memtoregPipe, set);
103    reg1 id_ex_reg_ALUsrc(ALUsrc, 1, latchID, ALUsrcPipe, set);
104    reg32 id_ex_reg_PC(PC_ID, 1, latchID, PC_EX, set);
```

```
105 //*****************************************************************************
106
107    buf #3 (req_to_ALU, req_to_EX); // DELAY FOR MUX (aluin2chooser)
108    mux32 aluin2chooser(data2pipe, immPipe, ALUsrcPipe, ALUinput2);
109    bundled2dual aluin1(data1, req_to_ALU, ALUinput1H, ALUinput1L);
110    bundled2dual aluin2(ALUinput2, req_to_ALU, ALUinput2H, ALUinput2L);
111    alu32 mainALU(ALUinput1H, ALUinput1L, ALUinput2H, ALUinput2L, realALUopPipe,
112                  ALUresultH, ALUresultL, nowhere3);
113    dual2bundled ALUdone(ALUresultH, ALUresultL, ALUresult, ALUreq);
114
115    memory MAIN_MEM(MEMresult, ALUresult, data2pipe, memwritePipe, memreadPipe,
116                    ALUreq, MEMreq);
117    mux32 resultchooser(ALUresult, MEMresult, memtoregPipe, writeD);
118    buf #3 (ack_from_EX, MEMreq); // DELAY FOR MUX (finalresultchooser)
119
120    assign pc = PC;
121    assign s1out = ALUresultH^ALUresultL;
122    assign s2in = ALUresultH; assign s2out = ALUresultL;
123    assign s3in = ALUinput2H; assign s3out = ALUinput2L;
124    assign temp = ALUreq;
125 endmodule // CPU
126
127 module main;
128    wire [31:0] pc, s1out, s2in, s2out, s3in, s3out, PC, PC_ID, PC_EX;
129    wire        temp;
130
131    CPU PipelinedAsync(pc, PC, PC_ID, PC_EX,
132                       s1out, s2in, s2out, s3in, s3out, temp);
133    initial
134      begin
135 //        $monitor("%d\nIF is at %d\nID is at %d\nEX is at %d\n",
136 //                 $time, PC, PC_ID, PC_EX);
137         /*$monitor(
138 "pc=%d\ns1ot=%b\ns2in=%b s2out=%b\ns3in=%b s3out=%b\ntemp=%b time=%d \n\n",
139                pc, s1out, s2in, s2out, s3in, s3out, temp, $time);*/
140         #8750 $stop;
141      end
142 endmodule // main
```

Appendix D

Laboratory Exercises for students

1. Using the single cycle asynchronous processor determine the delays of the global completion signal that are necessary to ensure correct operation.

2. Using the synchronous processor simulation and traditional boolean algebra convert the control unit to a simplified structural Verilog module. This lab is included because it is important for students to experience the difference between traditional logic design and the DIMS procedure.

3. Using the pipelined asynchronous processor convert the the behavioral control unit into a DIMS structural unit.

4. Using the pipelined asynchronous processor create a DIMS bit inverter to replace the behavioral bit inverter in the DIMS ALU. This requires that the control signals for the ALU be dual rail, which will be done in a later lab.

5. Using the pipelined asynchronous processor create a DIMS 4 input multiplexor to replace the behavioral multiplexor in the DIMS ALU. This also requires that the control signal be dual rail.

6. Using the pipelined asynchronous processor convert the remaining ALU controls to dual rail and integrate the changes from the previous labs.